THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 1A

A DRAFT OF SECTION 7.1.3: BITWISE TRICKS AND TECHNIQUES

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

Internet
Stanford GraphBase
MMIX

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also `http://www-cs-faculty.stanford.edu/~knuth/mmixware.html` for downloadable software to simulate the MMIX computer.

Copyright © 2006 by Addison–Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher, except that the official electronic file may be used to print single copies for personal (not commercial) use.

Zeroth printing (revision 2), 28 April 2007

STERNE
Internet
Stanford GraphBase

PREFACE

These unforeseen stoppages,
which I own I had no conception of when I first set out;
— but which, I am convinced now, will rather increase than diminish as I advance,
— have struck out a hint which I am resolved to follow;
— and that is, — not to be in a hurry;
— but to go on leisurely, writing and publishing two volumes of my life every year;
— which, if I am suffered to go on quietly, and can make a tolerable bargain
with my bookseller, I shall continue to do as long as I live.

— LAURENCE STERNE, *The Life and Opinions of*
Tristram Shandy, Gentleman (1760)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, and 3 were at the time of their first printings. And those carefully-checked volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make the text both interesting and authoritative, as far as it goes. But the field is vast; I cannot hope to have surrounded it enough to corral it completely. So I beg you to let me know about any deficiencies that you discover.

To put the material in context, this pre-fascicle contains Section 7.1.3 of a long, long chapter on combinatorial algorithms. Chapter 7 will eventually fill at least three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in the Stanford GraphBase, from which I will be drawing many examples. Then comes Section 7.1: Zeros and Ones, beginning with basic material about Boolean operations in Section 7.1.1 and Boolean evaluation in Section 7.1.2. Section 7.1.3, which you're about to read here, applies these ideas to make computer programs run fast. Section 7.1.4 will then discuss the representation of Boolean functions.

The next part, 7.2, is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns. Fascicles for this section have already appeared on the Web and/or in print. Section 7.2.2 will deal with backtracking in general. And so it will continue, if all goes well; an outline of

the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

This part of *The Art of Computer Programming* has probably been more fun to write than any other so far. Indeed, I've spent more than 30 years collecting material for Section 7.1.3; finally I'm able to assemble these goodies together and segue through them.

Most of Volume 4 will deal with abstract concepts, and there will be little or no need to say much about a computer's machine language. Volumes 1–3 have already dealt with most of the important ideas about programming at that level. But Section 7.1.3 is a notable exception: Here we often want to see the very pulse of the machine.

Therefore I strongly recommend that readers become familiar with the basics of the `MMIX` computer, explained in Volume 1 Fascicle 1, in order to fully appreciate the bitwise tricks and techniques described here. Cross-references to Sections 1.3.1′ and 1.3.2′ in the present booklet refer to that fascicle. I've reprinted the basic `MMIX` opcode-and-timing chart, Table 1.3.1′–1, at the end of this booklet for convenience, together with a list of ASCII codes.

The topic of Boolean functions and bit manipulation can of course be interpreted so broadly that it encompasses the entire subject of computer programming. The real goal of this fascicle is to focus on concepts that appear at the lowest levels, concepts on which we can erect significant superstructures. And even these apparently lowly notions turn out to be surprisingly rich, with explicit ties to sections 1.2.4, 1.2.5, 1.2.8, 2.3.1, 2.3.3, 2.3.4.2, 2.3.5, 3.1, 3.2.2, 4.1, 4.4, 4.5.3, 4.5.4, 4.6.1, 4.6.2, 4.6.3, 4.6.4, 5, 5.2.2, 5.2.3, 5.2.5, and 5.3.4 of the first three volumes. I strongly believe in building up a firm foundation, so I have discussed Boolean topics much more thoroughly than I will be able to do with material that is newer or less basic. Section 7.1.3 presented me with an extreme embarrassment of riches: After typing the manuscript I was astonished to discover that I had come up with 211 exercises, even though — believe it or not — I had to eliminate quite a lot of the interesting material that appears in my files.

My notes on combinatorial algorithms have been accumulating for more than forty years, so I fear that in several respects my knowledge is woefully behind the times. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 61, 76, 112, 117, 126, 128, 129, 130, and 174; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises 21, 140, 141, 156, and 165. Are those problems still open? Please inform me if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises 5, 6, 20, 26, 34, 39, 49, 50, 53, 57, 58(d,e), 59, 60, 72, 78, 80, 81, 82, 83,

84, 86, 90, 95, 110, 115, 116, 120, 121, 127, 146, 154, 155, 159, 168, 183, 193, and 198, and/or the answers to exercises 17, 18, and 139. Furthermore I've credited exercises 45 and 54 to unpublished work of Tom Rokicki and Bill Gosper. Have either of those results ever appeared in print, to your knowledge?

I shall happily pay a finder's fee of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00'; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
16 December 2006

[These techniques] are instances of general mathematical principles
waiting to be discovered, if an appropriate setting is created.
Such a setting would be a calculus of bitmap operations, so one can learn
to use these operations just as naturally as arithmetic operations on numbers.
— L. J. GUIBAS and J. STOLFI, *ACM Transactions on Graphics* (1982)

A nice mixture of boolean and numeric functions —
a suitable exercise for biturgical acolytes.
— R. W. GOSPER (1996)

A note on notation. Several formulas in Section 7.1.3 use the notation $\langle xyz \rangle$, for the median function (aka majority function) that is discussed extensively in Section 7.1.1. Other formulas use the notation $x \dot{-} y$, for the monus function (aka dot-minus or saturated subtraction), which was defined in Section 1.3.1′. Hexadecimal constants are preceded by a sharp sign: #123 means $(123)_{16}$. If you run across other notations that may be unfamiliar, please look at the Index to Notations at the end of Volumes 1, 2, or 3, and/or the entries under "Notation" in the index to the present booklet. Of course Volume 4 will some day contain its own Index to Notations.

Rokicki
Gosper
Knuth
GUIBAS
STOLFI
GOSPER
notation $\langle xyz \rangle$
median function
majority function
notation $x \dot{-} y$
monus function
dot-minus
saturated subtraction
Hexadecimal constants
Notation

Braymore,Caroline
Rochdale,Simon
COLMAN
bitwise–

Lady Caroline. *Psha! that's such a hack!*
Sir Simon. *A hack, Lady Caroline, that the knowing ones have warranted sound.*
— GEORGE COLMAN, *John Bull*, Act 3, Scene 1 (1803)

7.1.3. Bitwise Tricks and Techniques

Now comes the fun part: We get to use Boolean operations in our programs.

People are more familiar with arithmetic operations like addition, subtraction, and multiplication than they are with bitwise operations such as "and," "exclusive-or," and so on, because arithmetic has a very long history. But we will see that Boolean operations on binary numbers deserve to be much better known. Indeed, they're an important component of every good programmer's toolkit.

Early machine designers provided fullword bitwise operations in their computers primarily because such instructions could be included in a machine's repertoire almost for free. Binary logic seemed to be potentially useful, although

only a few applications were originally foreseen. For example, the EDSAC computer, completed in 1949, included a "collate" command that essentially performed the operation $z \leftarrow z + (x\&y)$, where z was the accumulator, x was the multiplier register, and y was a specified word in memory; it was used for unpacking data. The Manchester Mark I computer, built at about the same time, included not only bitwise AND, but also OR and XOR. When Alan Turing wrote its first programming manual in 1950, he remarked that bitwise NOT can be obtained by using XOR (denoted '$\not\equiv$') in combination with a row of 1s. R. A. Brooker, who extended Turing's manual in 1952 when the Mark II computer was being designed, remarked further that OR could be used "to round off a number by forcing 1 into its least significant digit position." By this time the Mark II, which was to become the prototype of the Ferranti Mercury, had also acquired new instructions for sideways addition and for the position of the most significant 1.

Keith Tocher published an unusual application of AND and OR in 1954, which has subsequently been reinvented frequently (see exercise 85). And during the ensuing decades, programmers have gradually discovered that bitwise operations can be amazingly useful. Many of these tricks have remained part of the folklore; the time is now ripe to take advantage of what has been learned.

A *trick* is a clever idea that can be used once, while a *technique* is a trick that can be used at least twice. We will see in this section that tricks tend to evolve naturally into techniques.

Enriched arithmetic. Let's begin by officially defining bitwise operations on integers so that, if $x = (\ldots x_2x_1x_0)_2$, $y = (\ldots y_2y_1y_0)_2$, and $z = (\ldots z_2z_1z_0)_2$ in binary notation, we have

$$x \mathbin{\&} y = z \iff x_k \wedge y_k = z_k, \quad \text{for all } k \ge 0; \tag{1}$$

$$x \mid y = z \iff x_k \vee y_k = z_k, \quad \text{for all } k \ge 0; \tag{2}$$

$$x \oplus y = z \iff x_k \oplus y_k = z_k, \quad \text{for all } k \ge 0. \tag{3}$$

(It would be tempting to write '$x \wedge y$' instead of $x\&y$, and '$x \vee y$' instead of $x|y$; but when we study optimization problems we'll find it better to reserve the notations $x \wedge y$ and $x \vee y$ for $\min(x,y)$ and $\max(x,y)$, respectively.) Thus, for example,

$$5 \mathbin{\&} 11 = 1, \qquad 5 \mid 11 = 15, \qquad \text{and} \qquad 5 \oplus 11 = 14,$$

since $5 = (0101)_2$, $11 = (1011)_2$, $1 = (0001)_2$, $15 = (1111)_2$, and $14 = (1110)_2$. Negative integers are to be thought of in this connection as infinite-precision numbers in two's complement notation, having infinitely many 1s at the left; for example, -5 is $(\ldots 1111011)_2$. Such infinite-precision numbers are a special case of 2-adic integers, which are discussed in exercise 4.1–31, and in fact the operators $\&$, $|$, $\oplus$ make perfect sense when they are applied to arbitrary 2-adic numbers.

Mathematicians have never paid much attention to the properties of $\&$ and $|$ as operations on integers. But the third operation, $\oplus$, has a venerable history, because it describes a winning strategy in the game of nim (see exercises 8–16). For this reason $x \oplus y$ has often been called the "nim sum" of the integers x and y.

EDSAC computer
collation, see bitwise and unpacking
Manchester Mark I computer
AND
OR
XOR
Turing
NOT
Brooker
Mark II computer (Manchester/Ferranti)
round off
Ferranti Mercury
sideways addition
most significant 1
Tocher
tricks versus techniques
infinite-precision numbers
two's complement notation
2-adic integers
nim
nim sum

All three of the basic bitwise operations turn out to have many useful properties. For example, every relation between $\wedge$, $\vee$, and $\oplus$ that we studied in Section 7.1.1 is automatically inherited by &, |, and $\oplus$ on integers, since the relation holds in every bit position. We might as well recap the main identities here:

commutative laws
associative laws
distributive laws
absorption laws
complementation
notation: $\sim x$
negation
subtraction
addition
shift binary

$$x \mathbin{\&} y = y \mathbin{\&} x, \qquad x \mid y = y \mid x, \qquad x \oplus y = y \oplus x; \tag{4}$$

$$(x \mathbin{\&} y) \mathbin{\&} z = x \mathbin{\&} (y \mathbin{\&} z), \quad (x \mid y) \mid z = x \mid (y \mid z), \quad (x \oplus y) \oplus z = x \oplus (y \oplus z); \tag{5}$$

$$(x \mid y) \mathbin{\&} z = (x \mathbin{\&} z) \mid (y \mathbin{\&} z), \qquad (x \mathbin{\&} y) \mid z = (x \mid z) \mathbin{\&} (y \mid z); \tag{6}$$

$$(x \oplus y) \mathbin{\&} z = (x \mathbin{\&} z) \oplus (y \mathbin{\&} z); \tag{7}$$

$$(x \mathbin{\&} y) \mid x = x, \qquad (x \mid y) \mathbin{\&} x = x; \tag{8}$$

$$(x \mathbin{\&} y) \oplus (x \mid y) = x \oplus y; \tag{9}$$

$$x \mathbin{\&} 0 = 0, \qquad x \mid 0 = x, \qquad x \oplus 0 = x; \tag{10}$$

$$x \mathbin{\&} x = x, \qquad x \mid x = x, \qquad x \oplus x = 0; \tag{11}$$

$$x \mathbin{\&} -1 = x, \qquad x \mid -1 = -1, \qquad x \oplus -1 = \bar{x}; \tag{12}$$

$$x \mathbin{\&} \bar{x} = 0, \qquad x \mid \bar{x} = -1, \qquad x \oplus \bar{x} = -1; \tag{13}$$

$$\overline{x \mathbin{\&} y} = \bar{x} \mid \bar{y}, \qquad \overline{x \mid y} = \bar{x} \mathbin{\&} \bar{y}, \qquad \overline{x \oplus y} = \bar{x} \oplus y = x \oplus \bar{y}. \tag{14}$$

The notation $\bar{x}$ in (12), (13), and (14) stands for bitwise *complementation* of x, namely $(\ldots \bar{x}_2\bar{x}_1\bar{x}_0)_2$, also written $\sim x$. Notice that (12) and (13) aren't quite the same as 7.1.1–(10) and 7.1.1–(18); we must now use $-1 = (\ldots 1111)_2$ instead of $1 = (\ldots 0001)_2$ in order to make the formulas bitwise correct.

We say that x is *contained in* y, written $x \subseteq y$ or $y \supseteq x$, if the individual bits of x and y satisfy $x_k \le y_k$ for all $k \ge 0$. Thus

$$x \subseteq y \iff x \mathbin{\&} y = x \iff x \mid y = y \iff x \mathbin{\&} \bar{y} = 0. \tag{15}$$

Of course we needn't use bitwise operations only in connection with each other; we can combine them with all the ordinary operations of arithmetic. For example, from the relation $x + \overline{x} = (\ldots 1111)_2 = -1$ we can deduce the formula

$$-x \;=\; \overline{x} + 1, \tag{16}$$

which turns out to be extremely important. Replacing x by $x - 1$ gives also

$$-x \;=\; \overline{x - 1}; \tag{17}$$

and in general we can reduce subtraction to complementation and addition:

$$\overline{x - y} \;=\; \overline{x} + y. \tag{18}$$

We often want to shift binary numbers to the left or right. These operations are equivalent to multiplication and division by powers of 2, but it is convenient to have special notations for them:

$$x \ll k \;=\; x \text{ shifted left } k \text{ bits} \;=\; \lfloor 2^k x \rfloor; \tag{19}$$

$$x \gg k \;=\; x \text{ shifted right } k \text{ bits} \;=\; \lfloor 2^{-k} x \rfloor. \tag{20}$$

Here k can be any integer, possibly negative. In particular we have

$$x \ll (-k) = x \gg k \qquad \text{and} \qquad x \gg (-k) = x \ll k, \tag{21}$$

for every infinite-precision number x. Also $(x \,\&\, y) \ll k = (x \ll k) \,\&\, (y \ll k)$, etc.

When bitwise operations are combined with addition, subtraction, multiplication, and/or shifting, extremely intricate results can arise, even when the formulas are quite short. A taste of the possibilities can be seen, for example, in Fig. 7. Furthermore, such formulas do not merely produce purposeless, chaotic behavior: A famous chain of operations known as "Gosper's hack," first published in 1972, opened people's eyes to the fact that a large number of useful and nontrivial functions can be computed rapidly (see exercise 20). Our goal in this section is to explore how such efficient constructions might be discovered.

infinite-precision
Sleator
quilt
pixel pattern
black
white
Gosper's hack
packing++
unpacking++
Lehmer
fractional precision
date
mod
division

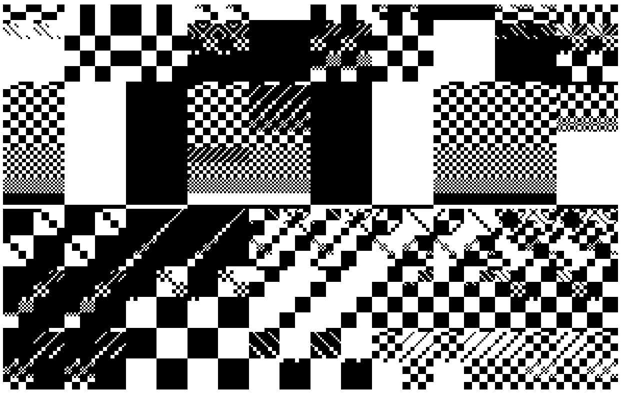


Fig. 11. A small portion of the patchwork quilt defined by the bitwise function $f(x,y) = ((x \oplus \bar{y}) \,\&\, ((y - 350) \gg 3))^2$; the square cell in column x and row y is painted white or black according as the value of $((f(x,y) \gg 12) \,\&\, 1)$ is 0 or 1. (Design by D. Sleator, 1976; see also exercise 18.)

Packing and unpacking. We studied algorithms for multiple-precision arithmetic in Section 4.3.1, dealing with situations where integers are too large to fit in a single word of memory or a single computer register. But the opposite situation, when integers are significantly *smaller* than the capacity of one computer word, is actually much more common; D. H. Lehmer called this "fractional precision." We can often deal with several integers at once, by packing them into a single word.

For example, a date x that consists of a year number y, a month number m, and a day number d, can be represented by using 4 bits for m and 5 bits for d:

$$x \;=\; (((y \ll 4) + m) \ll 5) + d. \tag{22}$$

We'll see below that many operations can be performed directly on dates in this packed form. For example, $x < x'$ when date x precedes date x'. But if necessary the individual components (y, m, d) can readily be unpacked when x is given:

$$d = x \bmod 32, \qquad m = (x \gg 5) \bmod 16, \qquad y = x \gg 9. \tag{23}$$

And these "mod" operations do not require division, because of the important law

$$x \bmod 2^n \;=\; x \,\&\, (2^n - 1). \tag{24}$$

For example, we have $d = x \,\&\, 31$ in (22) and (23).

Such packing of data obviously saves space in memory, and it also saves time: We can more quickly move or copy items of data from one place to another when

cache memory
prime numbers
table lookup by shifting
sieve of Eratosthenes

they've been packed together. Moreover, modern computers run considerably faster when they operate on numbers that fit into a cache memory of limited size.

The ultimate packing density is achieved when we have 1-bit items, because we can then cram 64 of them into a single 64-bit word. Suppose, for example, that we want a table of all odd prime numbers less than 1024, so that we can easily decide the primality of a small integer. No problem; only eight 64-bit numbers are required:

$$
\begin{aligned}
P_0 &= 0111011011010011001011010010011001011001010010001011011010000001,\\
P_1 &= 0100110000110010010100100110000110110000010000010110100110000100,\\
P_2 &= 1001001100101100001000000101101000000010010000110100100010010010 1,\\
P_3 &= 0010001010001000011000011001010010001011010000010001010001010010,\\
P_4 &= 0000110000000001001000010010011001000010010011001001011000001000 0,\\
P_5 &= 1101001001100000101001000100001000100001000100100101000100101000,\\
P_6 &= 1010000000100001000000110000110111000010000000101101000000101101 0000,\\
P_7 &= 0000010100010000100010100100100000010100100100010010000010100110.
\end{aligned}
$$

To test whether $2k+1$ is prime, for $0 \le k < 512$, we simply compute

$$P_{\lfloor k/64 \rfloor} \ll (k \mathbin{\&} 63) \tag{25}$$

in a 64-bit register, and see if the leftmost bit is 1. For example, the following `MMIX` instructions will do the job, if register `pbase` holds the address of P_0:

`SRU`	`$0,k,3`	$\$0 \leftarrow \lfloor k/8 \rfloor$ (i.e., $k \gg 3$).	
`LDOU`	`$1,pbase,$0`	$\$1 \leftarrow P_{\lfloor \$0/8 \rfloor}$.	
`AND`	`$0,k,#3f`	$\$0 \leftarrow k \bmod 64$ (i.e., k & `#3f`).	(26)
`SLU`	`$1,$1,$0`	$\$1 \leftarrow (\$1 \ll \$0) \bmod 2^{64}$.	
`BN`	`$1,PRIME`	Branch to `PRIME` if $s(\$1) < 0$. ▮	

Notice that the leftmost bit of a register is 1 if and only if the register contents are negative.

We could equally well pack the bits from right to left in each word:

$$
\begin{aligned}
Q_0 &= 1000000101101101000100101001101001100100101101001100101101101110,\\
Q_1 &= 0010000110010110100000100000110110000110010010100100110000110010,\\
Q_2 &= 1010010010001001011000010010000000101101000000010000110100110010 01,\\
Q_3 &= 0100101000101000100000010110100010010100110000110000100010100010 0,\\
Q_4 &= 0000100000011010010011001001000010011001001000010010000000001100 00,\\
Q_5 &= 0001010010001010010010001000010001000010001001010000001100100101 1,\\
Q_6 &= 0000101101000000010110100000001000011011000011000000100001000000 0101,\\
Q_7 &= 0110010100000100100010010010100000010010010100010000100010100000;
\end{aligned}
$$

here $Q_j = P_j^R$. Instead of shifting left as in (25), we now shift right,

$$Q_{\lfloor k/64 \rfloor} \gg (k \mathbin{\&} 63), \tag{27}$$

and look at the *rightmost* bit of the result. The last two lines of (26) become

`SRU`	`$1,$1,$0`	$\$1 \leftarrow \$1 \gg \$0$.	(28)
`BOD`	`$1,PRIME`	Branch to `PRIME` if $1 is odd. ▮	

(And of course we use `qbase` instead of `pbase`.) Either way, the classic *sieve of Eratosthenes* will readily set up the basic table entries P_j or Q_j (see exercise 24).

big-endian++
little-endian++
multiple-precision

Table 1

THE BIG-ENDIAN VIEW OF A 32-BYTE MEMORY

Octa	Tetra	Wyde	Byte	Bits
octa 0	tetra 0	wyde 0	byte 0	$a_0 \ldots a_7$
			byte 1	$a_8 \ldots a_{15}$
		wyde 2	byte 2	$a_{16} \ldots a_{23}$
			byte 3	$a_{24} \ldots a_{31}$
	tetra 4	wyde 4	byte 4	$a_{32} \ldots a_{39}$
			byte 5	$a_{40} \ldots a_{47}$
		wyde 6	byte 6	$a_{48} \ldots a_{55}$
			byte 7	$a_{56} \ldots a_{63}$
octa 8	tetra 8	wyde 8	byte 8	$a_{64} \ldots a_{71}$
			byte 9	$a_{72} \ldots a_{79}$
		wyde 10	byte 10	$a_{80} \ldots a_{87}$
			byte 11	$a_{88} \ldots a_{95}$
	tetra 12	wyde 12	byte 12	$a_{96} \ldots a_{103}$
			byte 13	$a_{104} \ldots a_{111}$
		wyde 14	byte 14	$a_{112} \ldots a_{119}$
			byte 15	$a_{120} \ldots a_{127}$
octa 16	tetra 16	wyde 16	byte 16	$a_{128} \ldots a_{135}$
			byte 17	$a_{136} \ldots a_{143}$
		wyde 18	byte 18	$a_{144} \ldots a_{151}$
			byte 19	$a_{152} \ldots a_{159}$
	tetra 20	wyde 20	byte 20	$a_{160} \ldots a_{167}$
			byte 21	$a_{168} \ldots a_{175}$
		wyde 22	byte 22	$a_{176} \ldots a_{183}$
			byte 23	$a_{184} \ldots a_{191}$
octa 24	tetra 24	wyde 24	byte 24	$a_{192} \ldots a_{199}$
			byte 25	$a_{200} \ldots a_{207}$
		wyde 26	byte 26	$a_{208} \ldots a_{215}$
			byte 27	$a_{216} \ldots a_{223}$
	tetra 28	wyde 28	byte 28	$a_{224} \ldots a_{231}$
			byte 29	$a_{232} \ldots a_{239}$
		wyde 30	byte 30	$a_{240} \ldots a_{247}$
			byte 31	$a_{248} \ldots a_{255}$

Big-endian and little-endian conventions. Whenever we pack bits or bytes into words, we must decide whether to place them from left to right or from right to left. The left-to-right convention is called "big-endian," because the initial items go into the most significant positions; thus they will have bigger significance than their successors, when numbers are compared. The right-to-left convention is called "little-endian"; it puts the first items where little numbers go.

A big-endian approach seems more natural in many cases, because we're accustomed to reading and writing from left to right. But a little-endian placement has advantages too. For example, let's consider the prime number problem again; let $a_k = [2k{+}1 \text{ is prime}]$. Our table entries $\{P_0, P_1, \ldots, P_7\}$ are big-endian, and we can regard them as the representation of a single multiple-precision integer that is 512 bits long:

$$(P_0P_1 \ldots P_7)_{2^{64}} \;=\; (a_0a_1 \ldots a_{511})_2. \tag{29}$$

Similarly, our little-endian table entries represent the multiprecise integer

$$(Q_7 \ldots Q_1Q_0)_{2^{64}} \;=\; (a_{511} \ldots a_1a_0)_2. \tag{30}$$

The latter integer is mathematically nicer than the former, because it is

$$\sum_{k=0}^{511} 2^k a_k = \sum_{k=0}^{511} 2^k[2k{+}1 \text{ is prime}] = \Bigl(\sum_{k=0}^{\infty} 2^k[2k{+}1 \text{ is prime}]\Bigr) \bmod 2^{512}. \tag{31}$$

portability+

Table 2

THE LITTLE-ENDIAN VIEW OF A 32-BYTE MEMORY

octa 24							
tetra 28				tetra 24			
wyde 30		wyde 28		wyde 26		wyde 24	
byte 31	byte 30	byte 29	byte 28	byte 27	byte 26	byte 25	byte 24
$a_{255}\ldots a_{248}$	$a_{247}\ldots a_{240}$	$a_{239}\ldots a_{232}$	$a_{231}\ldots a_{224}$	$a_{223}\ldots a_{216}$	$a_{215}\ldots a_{208}$	$a_{207}\ldots a_{200}$	$a_{199}\ldots a_{192}$
octa 16							
tetra 20				tetra 16			
wyde 22		wyde 20		wyde 18		wyde 16	
byte 23	byte 22	byte 21	byte 20	byte 19	byte 18	byte 17	byte 16
$a_{191}\ldots a_{184}$	$a_{183}\ldots a_{176}$	$a_{175}\ldots a_{168}$	$a_{167}\ldots a_{160}$	$a_{159}\ldots a_{152}$	$a_{151}\ldots a_{144}$	$a_{143}\ldots a_{136}$	$a_{135}\ldots a_{128}$
octa 8							
tetra 12				tetra 8			
wyde 14		wyde 12		wyde 10		wyde 8	
byte 15	byte 14	byte 13	byte 12	byte 11	byte 10	byte 9	byte 8
$a_{127}\ldots a_{120}$	$a_{119}\ldots a_{112}$	$a_{111}\ldots a_{104}$	$a_{103}\ldots a_{96}$	$a_{95}\ldots a_{88}$	$a_{87}\ldots a_{80}$	$a_{79}\ldots a_{72}$	$a_{71}\ldots a_{64}$
octa 0							
tetra 4				tetra 0			
wyde 6		wyde 4		wyde 2		wyde 0	
byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
$a_{63}\ldots a_{56}$	$a_{55}\ldots a_{48}$	$a_{47}\ldots a_{40}$	$a_{39}\ldots a_{32}$	$a_{31}\ldots a_{24}$	$a_{23}\ldots a_{16}$	$a_{15}\ldots a_{8}$	$a_{7}\ldots a_{0}$

Notice, however, that we used $(Q_7\ldots Q_1Q_0)_{2^{64}}$ to get this simple result, not $(Q_0Q_1\ldots Q_7)_{2^{64}}$. The other number,

$$(Q_0Q_1\ldots Q_7)_{2^{64}} = (a_{63}\ldots a_1a_0a_{127}\ldots a_{65}a_{64}a_{191}\ldots a_{385}a_{384}a_{511}\ldots a_{449}a_{448})_2$$

is in fact quite weird, and it has no really nice formula. (See exercise 25.)

Endianness has important consequences, because most computers allow individual bytes of the memory to be addressed as well as register-sized units. `MMIX` has a big-endian architecture; therefore if register `x` contains the 64-bit number #`0123456789abcdef`, and if we use the commands '`STOU x,0; LDBU y,1`' to store `x` into octabyte location 0 and read back the byte in location 1, the result in register `y` will be #`23`. On machines with a little-endian architecture, the analogous commands would set `y` $\leftarrow$ #`cd` instead; #`23` would be byte 6.

Tables 1 and 2 illustrate the competing "world views" of big-endian and little-endian aficionados. The big-endian approach is basically top-down, with bit 0 and byte 0 at the top left; the little-endian approach is basically bottom-up, with bit 0 and byte 0 at the bottom right. Because of this difference, great care is necessary when transmitting data from one kind of computer to another, or when writing programs that are supposed to give equivalent results in both cases. On the other hand, our example of the Q table for primes shows that we can perfectly well use a little-endian packing convention on a big-endian computer

like `MMIX`, or vice versa. The difference is noticeable only when data is loaded and stored in different-sized chunks, or passed between machines.

Working with the rightmost bits. Big-endian and little-endian approaches aren't readily interchangeable in general, because the laws of arithmetic send signals leftward from the bits that are "least significant." Some of the most important bitwise manipulation techniques are based on this fact.

If x is almost any nonzero 2-adic integer, we can write its bits in the form

$$x \;=\; (\alpha\, 01^a 10^b)_2; \tag{32}$$

in other words, x consists of some arbitrary (but infinite) binary string α, followed by a 0, which is followed by $a+1$ ones, and followed by b zeros, for some $a \ge 0$ and $b \ge 0$. (The exceptions are when $x = -2^b$ and $a = \infty$.) Consequently

$$\bar{x} \;=\; (\bar{\alpha}\, 10^a 01^b)_2, \tag{33}$$
$$x-1 \;=\; (\alpha\, 01^a 01^b)_2, \tag{34}$$
$$-x \;=\; (\bar{\alpha}\, 10^a 10^b)_2; \tag{35}$$

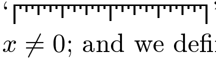
and we see that $\bar{x}+1 = -x = \overline{x-1}$, in agreement with (16) and (17). With two operations we can therefore compute relatives of x in several useful ways:

$$x \mathbin{\&} (x-1) = (\ \alpha\ 01^a 00^b)_2 \quad \text{[remove the rightmost 1]}; \tag{36}$$
$$x \mathbin{\&} -x = (0^\infty 00^a 10^b)_2 \quad \text{[extract the rightmost 1]}; \tag{37}$$
$$x \mid -x = (1^\infty 11^a 10^b)_2 \quad \text{[smear the rightmost 1 to the left]}; \tag{38}$$
$$x \oplus -x = (1^\infty 11^a 00^b)_2 \quad \text{[remove and smear it to the left]}; \tag{39}$$
$$x \mid (x-1) = (\ \alpha\ 01^a 11^b)_2 \quad \text{[smear the rightmost 1 to the right]}; \tag{40}$$
$$x \oplus (x-1) = (0^\infty 00^a 11^b)_2 \quad \text{[extract and smear it to the right]}; \tag{41}$$
$$\bar{x} \mathbin{\&} (x-1) = (0^\infty 00^a 01^b)_2 \quad \text{[extract, remove, and smear it to the right]}. \tag{42}$$

And two further operations produce yet another variant:

$$((x|(x-1))+1) \mathbin{\&} x = (\ \alpha\ 00^a 00^b)_2 \quad \text{[remove the rightmost run of 1s]}. \tag{43}$$

When $x = 0$, half of these formulas produce 0, the other half give -1. [Formula (36) is due to Peter Wegner, *CACM* **3** (1960), 322; and (43) is due to H. Tim Gladwin, *CACM* **14** (1971), 407–408. See also Henry S. Warren, Jr., *CACM* **20** (1977), 439–441.]

The quantity b in these formulas, which specifies the number of trailing zeros in x, is called the *ruler function* of x and written ρx, because it is related to the lengths of the tick marks that are often used to indicate fractions of an inch: ‘’. In general, ρx is the highest power of 2 that divides x, when $x \ne 0$; and we define $\rho 0 = \infty$. The recurrence relations

$$\rho(2x+1) = 0, \qquad \rho(2x) = \rho(x) + 1 \tag{44}$$

also serve to define ρx for nonzero x. Another handy relation is worthy of note,

$$\rho(x-y) \;=\; \rho(x \oplus y). \tag{45}$$

rightmost bits++
smearing bits
extracting bits
removing bits
runs of bits
Wegner
Gladwin
Warren
trailing zeros
ruler function
ρx
binary valuation, see ruler function

The elegant formula $x \mathbin{\&} -x$ in (37) allows us to *extract* the rightmost 1 bit very nicely, but we often want to identify exactly which bit it is. The ruler function can be computed in many ways, and the best method often depends heavily on the computer that is being used. For example, a two-instruction sequence due to J. Dallos does the job quickly and easily on MMIX:

```
SUBU t,x,1;  SADD rho,t,x.
```
(46)

We shall discuss here two approaches that do not rely on exotic commands like SADD; and later, after learning a few more techniques, we'll consider a third way.

The first general-purpose method makes use of "magic mask" constants μ_k that prove to be useful in many other applications, namely

$$\begin{aligned} \mu_0 &= (\ldots 101010101010101010101010101010101)_2 = -1/3,\\ \mu_1 &= (\ldots 100110011001100110011001100110011)_2 = -1/5,\\ \mu_2 &= (\ldots 100001111000011110000111100001111)_2 = -1/17, \end{aligned} \tag{47}$$

and so on. In general μ_k is the infinite 2-adic fraction $-1/(2^{2^k}+1)$, because $(2^{2^k}+1)\mu_k = (\mu_k \ll 2^k) + \mu_k = (\ldots 11111)_2 = -1$. On a computer that has 2^d-bit registers we don't need infinite precision, of course, so we use the truncated constants

$$\mu_{d,k} = (2^{2^d}-1)/(2^{2^k}+1) \qquad \text{for } 0 \le k < d. \tag{48}$$

These constants are familiar from our study of Boolean evaluation, because they are the truth tables of the projection functions x_{n-k} (see, for example, 7.1.2–(7)).

When x is a power of 2, we can use these masks to compute

$$\rho x = [x \mathbin{\&} \mu_0 = 0] + 2[x \mathbin{\&} \mu_1 = 0] + 4[x \mathbin{\&} \mu_2 = 0] + 8[x \mathbin{\&} \mu_3 = 0] + \cdots, \tag{49}$$

because $[2^j \mathbin{\&} \mu_k = 0] = j_k$ when $j = (\ldots j_3 j_2 j_1 j_0)_2$. Thus, on a 2^d-bit computer, we can start with $\rho \leftarrow 0$ and $y \leftarrow x \mathbin{\&} -x$; then set $\rho \leftarrow \rho + 2^k$ if $y \mathbin{\&} \mu_{d,k} = 0$, for $0 \le k < d$. This procedure gives $\rho = \rho x$ when $x \ne 0$. (It also gives $\rho 0 = 2^d - 1$, an anomalous value that may need to be corrected; see exercise 30.)

For example, the corresponding MMIX program might look like this:

```
m0 GREG #5555555555555555  ;m1 GREG #3333333333333333;
m2 GREG #0f0f0f0f0f0f0f0f  ;m3 GREG #00ff00ff00ff00ff;
m4 GREG #0000ffff0000ffff  ;m5 GREG #00000000ffffffff;
 NEGU y,x;  AND y,x,y;  AND q,y,m5; ZSZ rho,q,32;
 AND q,y,m4;  ADD t,rho,16;  CSZ rho,q,t;
 AND q,y,m3;  ADD t,rho,8;  CSZ rho,q,t;
 AND q,y,m2;  ADD t,rho,4;  CSZ rho,q,t;
 AND q,y,m1;  ADD t,rho,2;  CSZ rho,q,t;
 AND q,y,m0;  ADD t,rho,1;  CSZ rho,q,t;
```
(50)

total time $= 19\upsilon$. Or we could replace the last three lines by

```
SRU y,y,rho;  LDB t,rhotab,y;  ADD rho,rho,t
```
(51)

where rhotab points to the beginning of an appropriate 129-byte table. The total time would then be $\mu + 13\upsilon$.

Dallos
SADD
magic mask
mask: A bit pattern with 1s in key position
2-adic fraction
truth tables
projection functions
MMIX
CSZ
ZSZ

The second general-purpose approach to the computation of ρx is quite different. On a 64-bit machine it starts as before, with $y \leftarrow x \mathbin{\&} -x$; but then it simply sets

$$\rho \;\leftarrow\; decode\bigl[((a \cdot y) \bmod 2^{64}) \gg 58\bigr], \tag{52}$$

where a is a suitable multiplier and *decode* is a suitable 64-byte table. The constant $a = (a_{63} \ldots a_1 a_0)_2$ must have the property that its 64 substrings

$$a_{63}a_{62}\ldots a_{58},\; a_{62}a_{61}\ldots a_{57},\; \ldots,\; a_5a_4\ldots a_0,\; a_4a_3a_2a_1a_0 0,\; \ldots,\; a_0 00000$$

are distinct. Exercise 2.3.4.2–23 shows that many such "de Bruijn cycles" exist; for example, we can use M. H. Martin's constant `#03f79d71b4ca8b09`, which is discussed in exercise 3.2.2–17. The decoding table $decode[0], \ldots, decode[63]$ is then

$$\begin{array}{l} 00,01,56,02,57,49,28,03,61,58,42,50,38,29,17,04,\\ 62,47,59,36,45,43,51,22,53,39,33,30,24,18,12,05,\\ 63,55,48,27,60,41,37,16,46,35,44,21,52,32,23,11,\\ 54,26,40,15,34,20,31,10,25,14,19,09,13,08,07,06. \end{array} \tag{53}$$

[This technique was devised in 1997 by M. Läuter, and independently by C. E. Leiserson, H. Prokop, and K. H. Randall a few months later (unpublished). David Seal had used a similar method in 1994, with a larger decoding table.]

Working with the leftmost bits. The function $\lambda x = \lfloor \lg x \rfloor$, which is dual to ρx because it locates the *leftmost* 1 when $x > 0$, was introduced in Eq. 4.6.3–(6). It satisfies the recurrence

$$\lambda 1 = 0; \qquad \lambda(2x) = \lambda(2x+1) = \lambda(x) + 1 \quad \text{for } x > 0; \tag{54}$$

and it is undefined when $x \le 0$. What is a good way to compute it? Once again `MMIX` provides a quick-but-tricky solution:

```
FLOTU y,ROUND_DOWN,x; SUB y,y,fone; SR lam,y,52          (55)
```

where `fone` = `#3ff0000000000000` is the floating-point representation of 1.0. (Total time $6v$.) This code floats x, then extracts the exponent.

But if floating-point conversion is not readily available, a binary reduction strategy works fairly well on a 2^d-bit machine. We can start with $\lambda \leftarrow 0$ and $y \leftarrow x$; then we set $\lambda \leftarrow \lambda + 2^k$ and $y \leftarrow y \gg 2^k$ if $y \mathbin{\&} \bar{\mu}_k \ne 0$, for $k = d-1$, …, 1, 0 (or until k is reduced to the point where a short table can be used to finish up). The `MMIX` code analogous to (50) and (51) is now

```
ANDN q,x,m5; SRU z,x,32; SET y,x; CSNZ y,q,z; ZSNZ lam,q,32;
ANDN q,y,m4; SRU z,y,16; ADD t,lam,16; CSNZ y,q,z; CSNZ lam,q,t;
ANDN q,y,m3; SRU z,y,8; ADD t,lam,8; CSNZ y,q,z; CSNZ lam,q,t;
LDB t,lamtab,y; ADD lam,lam,t;                                    (56)
```

and the total time is $\mu + 17v$. In this case table `lamtab` has 256 entries, namely λx for $0 \le x < 256$. Notice that the "conditional set" (`CS`) and "zero or set" (`ZS`) instructions have been used here instead of branch instructions. They tend to save time, even though they've made the program slightly longer.

de Bruijn cycles
Martin
Läuter
Leiserson
Prokop
Randall
Seal
leftmost bits+
λx+
[lg x]+
binary logarithm+
leftmost
floating-point
CSNZ
ZSNZ
MMIX
conditional set
zero or set
branch instructions

There appears to be no simple way to extract the leftmost 1 bit that appears in a register, analogous to the trick by which we extracted the rightmost 1 in (37). For this purpose we could compute $y \leftarrow \lambda x$ and then $1 \ll y$, if $x \neq 0$; but a binary "smearing right" method is somewhat shorter and faster:

$$\begin{aligned}&\text{Set } y \leftarrow x, \text{ then } y \leftarrow y \mid (y \gg 2^k) \text{ for } 0 \leq k < d.\\ &\text{The leftmost 1 bit of } x \text{ is then } y - (y \gg 1).\end{aligned} \tag{57}$$

[These non-floating-point methods have been suggested by H. S. Warren, Jr.]

Other operations at the left of a register, like removing the leftmost run of 1s, are harder yet; see exercise 39. But there is a remarkably simple, machine-independent way to determine whether or not $\lambda x = \lambda y$, given unsigned integers x and y, in spite of the fact that we can't compute λx or λy quickly:

$$\lambda x = \lambda y \qquad \text{if and only if} \qquad x \oplus y \leq x \,\&\, y. \tag{58}$$

[See exercise 40. This elegant relation was discovered by W. C. Lynch in 2006.] We will use (58) below, to devise another way to compute λx.

Sideways addition. Binary n-bit numbers $x = (x_{n-1} \ldots x_1 x_0)_2$ are often used to represent subsets X of the n-element universe $\{0, 1, \ldots, n-1\}$, with $k \in X$ if and only if $2^k \subseteq x$. The functions λx and ρx then represent the largest and smallest elements of X. The function

$$\nu x \;=\; x_{n-1} + \cdots + x_1 + x_0, \tag{59}$$

which is called the "sideways sum" or "population count" of x, also has obvious importance in this connection, because it represents the cardinality $|X|$, namely the number of elements in X. This function, which we considered in 4.6.3–(7), satisfies the recurrence

$$\nu 0 = 0; \qquad \nu(2x) = \nu(x) \quad \text{and} \quad \nu(2x{+}1) = \nu(x) + 1, \quad \text{for } x \geq 0. \tag{60}$$

It also has an interesting connection with the ruler function (exercise 1.2.5–11),

$$\rho x = 1 + \nu(x{-}1) - \nu x; \qquad \text{equivalently,} \quad \sum_{k=1}^{n} \rho k = n - \nu n. \tag{61}$$

The first textbook on programming, *The Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Gill, second edition (Reading, Mass.: Addison–Wesley, 1957), 155, 191–193, presented an interesting subroutine for sideways addition due to D. B. Gillies and J. C. P. Miller. Their method was devised for the 35-bit numbers of the EDSAC, but it is readily converted to the following 64-bit procedure for νx when $x = (x_{63} \ldots x_1 x_0)_2$:

$$\begin{aligned}&\text{Set } y \leftarrow x - ((x \gg 1) \,\&\, \mu_0). \text{ (Now } y = (u_{31} \ldots u_1 u_0)_4, \text{ where } u_j = x_{j+1} + x_j.)\\ &\text{Set } y \leftarrow (y \,\&\, \mu_1) + ((y \gg 2) \,\&\, \mu_1). \text{ (Now } y = (v_{15} \ldots v_1 v_0)_{16}, \ v_j = u_{j+1} + u_j.)\\ &\text{Set } y \leftarrow (y + (y \gg 4)) \,\&\, \mu_2. \text{ (Now } y = (w_7 \ldots w_1 w_0)_{256}, \ w_j = v_{j+1} + v_j.)\\ &\text{Finally } \nu \leftarrow ((a \cdot y) \bmod 2^{64}) \gg 56, \text{ where } a = (11111111)_{256}.\end{aligned} \tag{62}$$

The last step cleverly computes $y \bmod 255 = w_7 + \cdots + w_1 + w_0$ via multiplication, using the fact that the sum fits comfortably in eight bits. [David Muller had programmed a similar method for the ILLIAC I machine in 1954.]

smearing right
Warren
run of 1s
Lynch
sum of bits, see sideways sum
ones counting, see sideways
sideways addition+
subsets
largest
smallest
population count
cardinality
Wilkes
Wheeler
Gill
Gillies
Miller
EDSAC
remainder mod $2^n - 1$
Muller
ILLIAC I

If x is expected to be "sparse," having at most a few 1 bits, we can use a faster method [P. Wegner, *CACM* **3** (1960), 322]:

$$\text{Set } \nu \leftarrow 0,\ y \leftarrow x. \text{ Then while } y \neq 0, \text{ set } \nu \leftarrow \nu + 1,\ y \leftarrow y \,\&\, (y-1). \tag{63}$$

A similar approach, using $y \leftarrow y \,|\, (y+1)$, works when x is expected to be "dense."

Bit reversal. For our next trick, let's change $x = (x_{63} \ldots x_1 x_0)_2$ to its left-right mirror image, $x^R = (x_0 x_1 \ldots x_{63})_2$. Anybody who has been following the developments so far, seeing methods like (50), (56), (57), and (62), will probably think, "Aha — once again we can divide by 2 and conquer! If we've already discovered how to reverse 32-bit numbers, we can reverse 64-bit numbers almost as fast, because $(xy)^R = y^R x^R$. All we have to do is apply the 32-bit method in parallel to both halves of the register, then swap the left half with the right half."

Right. For example, we can reverse an 8-bit string in three easy steps:

$$\begin{array}{ll} \text{Given} & x_7x_6x_5x_4x_3x_2x_1x_0 \\ \text{Swap bits} & x_6x_7x_4x_5x_2x_3x_0x_1 \\ \text{Swap nyps} & x_4x_5x_6x_7x_0x_1x_2x_3 \\ \text{Swap nybbles} & x_0x_1x_2x_3x_4x_5x_6x_7 \end{array} \tag{64}$$

And six such easy steps will reverse 64 bits. Fortunately, each of the swapping operations turns out to be quite simple with the help of the magic masks μ_k:

$$\begin{array}{l} y \leftarrow (x \gg 1) \,\&\, \mu_0, \quad z \leftarrow (x \,\&\, \mu_0) \ll 1, \quad x \leftarrow y \mid z; \\ y \leftarrow (x \gg 2) \,\&\, \mu_1, \quad z \leftarrow (x \,\&\, \mu_1) \ll 2, \quad x \leftarrow y \mid z; \\ y \leftarrow (x \gg 4) \,\&\, \mu_2, \quad z \leftarrow (x \,\&\, \mu_2) \ll 4, \quad x \leftarrow y \mid z; \\ y \leftarrow (x \gg 8) \,\&\, \mu_3, \quad z \leftarrow (x \,\&\, \mu_3) \ll 8, \quad x \leftarrow y \mid z; \\ y \leftarrow (x \gg 16) \,\&\, \mu_4, \quad z \leftarrow (x \,\&\, \mu_4) \ll 16, \quad x \leftarrow y \mid z; \\ x \leftarrow (x \gg 32) \mid ((x \ll 32) \bmod 2^{64}). \end{array} \tag{65}$$

[Christopher Strachey foresaw some aspects of this construction in *CACM* **4** (1961), 146, and a similar *ternary* method was devised in 1973 by Bruce Baumgart (see exercise 49). The mature algorithm (65) was presented by Henry S. Warren, Jr., in *Hacker's Delight* (Addison–Wesley, 2002), 102.]

But MMIX is once again able to trump this general-purpose technique with less traditional commands that do the job much faster. Consider

```
rev GREG #0102040810204080; MOR x,x,rev; MOR x,rev,x;        (66)
```

the first MOR instruction reverses the bytes of x from big-endian to little-endian or vice versa, while the second reverses the bits within each byte.

Bit swapping. Suppose we only want to interchange two bits within a register, $x_i \leftrightarrow x_j$, where $i > j$. What would be a good way to proceed? (Dear reader, please pause for a moment and solve this problem in your head, or with pencil and paper — without looking at the answer below.)

Let $\delta = i - j$. Here is one solution (but don't peek until you're ready):

$$y \leftarrow (x \gg \delta) \,\&\, 2^j,\ z \leftarrow (x \,\&\, 2^j) \ll \delta,\ x \leftarrow (x \,\&\, m) \mid y \mid z, \text{ where } \overline{m} = 2^i | 2^j. \tag{67}$$

Wegner
reversal of bits+
divide by 2 and conquer
magic masks
Strachey
Baumgart
Warren
big-endian
little-endian
MOR
swapping bits+++

It uses two shifts and five bitwise Boolean operations, assuming that i and j are given constants. It is like each of the first lines of (65), except that a new mask m is needed because y and z don't account for all of the bits of x.

We can, however, do better, saving one operation and one constant:

$$y \leftarrow (x \oplus (x \gg \delta)) \mathbin{\&} 2^j, \qquad x \leftarrow x \oplus y \oplus (y \ll \delta). \tag{68}$$

The first assignment now puts $x_i \oplus x_j$ into position j; the second changes x_i to $x_i \oplus (x_i \oplus x_j)$ and x_j to $x_j \oplus (x_i \oplus x_j)$, as desired. In general it's often wise to convert a problem of the form "change x to $f(x)$" into a problem of the form "change x to $x \oplus g(x)$," since the bit-difference $g(x)$ might be easy to calculate.

On the other hand, there's a sense in which (67) might be preferable to (68), because the assignments to y and z in (67) can sometimes be performed simultaneously. When expressed as a circuit, (67) has a depth of 4 while (68) has depth 5.

Operation (68) can of course be used to swap several pairs of bits simultaneously, when we use a general mask θ instead of 2^j:

$$y \leftarrow (x \oplus (x \gg \delta)) \mathbin{\&} \theta, \qquad x \leftarrow x \oplus y \oplus (y \ll \delta). \tag{69}$$

Let us call this operation a "δ-swap," because it allows us to swap any nonoverlapping pairs of bits that are δ places apart. The mask θ has a 1 in the rightmost position of each pair that's supposed to be swapped. For example, (69) will swap the leftmost 25 bits of a 64-bit word with the rightmost 25 bits, while leaving the 14 middle bits untouched, if we let $\delta = 39$ and $\theta = 2^{25} - 1 = \texttt{\#1ffffff}$.

Indeed, there's an astonishing way to reverse 64 bits using δ-swaps, namely

$$\begin{aligned}
&y \leftarrow (x \gg 1) \mathbin{\&} \mu_0, \quad z \leftarrow (x \mathbin{\&} \mu_0) \ll 1, \quad x \leftarrow y \mid z,\\
&y \leftarrow (x \oplus (x \gg 4)) \mathbin{\&} \texttt{\#0300c0303030c303}, \quad x \leftarrow x \oplus y \oplus (y \ll 4),\\
&y \leftarrow (x \oplus (x \gg 8)) \mathbin{\&} \texttt{\#00c0300c03f0003f}, \quad x \leftarrow x \oplus y \oplus (y \ll 8),\\
&y \leftarrow (x \oplus (x \gg 20)) \mathbin{\&} \texttt{\#00000ffc00003fff}, \quad x \leftarrow x \oplus y \oplus (y \ll 20),\\
&x \leftarrow (x \gg 34) \mid ((x \ll 30) \bmod 2^{64}),
\end{aligned} \tag{70}$$

saving two of the bitwise operations in (65) even though (65) looks "optimum."

***Bit permutation in general.** The methods we've just seen can be extended to obtain an *arbitrary* permutation of the bits in a register. In fact, there always exist masks $\theta_0, \ldots, \theta_5, \hat\theta_4, \ldots, \hat\theta_0$ such that the following operations transform $x = (x_{63} \ldots x_1 x_0)_2$ into any desired rearrangement $x^\pi = (x_{63\pi} \ldots x_{1\pi} x_{0\pi})_2$ of its bits:

$$\begin{aligned}
&x \leftarrow 2^k\text{-swap of } x \text{ with mask } \theta_k, \text{ for } k = 0, 1, 2, 3, 4, 5;\\
&x \leftarrow 2^k\text{-swap of } x \text{ with mask } \hat\theta_k, \text{ for } k = 4, 3, 2, 1, 0.
\end{aligned} \tag{71}$$

In general, a permutation of 2^d bits can be achieved with $2d - 1$ such steps, using appropriate masks θ_k, $\hat\theta_k$, where the swap distances are respectively 2^0, 2^1, …, 2^{d-1}, …, 2^1, 2^0.

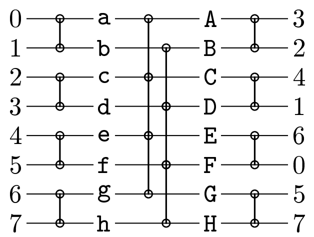
To prove this fact, we can use a special case of the permutation networks discovered independently by A. M. Duguid and J. Le Corre in 1959, based on earlier work of D. Slepian [see V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic* (New York: Academic Press, 1965), Section 3.3].

depth
δ-swap
bit permutation++++
permutation networks
Duguid
Le Corre
Slepian
Beneš

rearrangeable networks, see perm networks
crossbar module
graph
bipartite graph

Figure 12 shows a permutation network $P(2n)$ for $2n$ elements constructed from two permutation networks for n elements, when $n = 4$. Each ‘|’ connection between two lines represents a *crossbar module* that either leaves the line contents unaltered or interchanges them, as the data flows from left to right. Every setting of the individual crossbars therefore causes $P(2n)$ to produce a permutation of its inputs; conversely, we wish to show that any permutation of the $2n$ inputs can be achieved by some setting of the crossbars.

The construction of Fig. 12 is best understood by considering an example. Suppose we want to route the inputs $(0,1,2,3,4,5,6,7)$ to $(3,2,4,1,6,0,5,7)$, respectively. The first job is to determine the contents of the lines just after the first column of crossbars and just before the last column, since we can then use a similar method to set the crossbars in the inner $P(4)$'s. Thus, in the network



(72)

we want to find permutations $\mathtt{abcdefgh}$ and $\mathtt{ABCDEFGH}$ such that $\{\mathtt{a},\mathtt{b}\} = \{0,1\}$, $\{\mathtt{c},\mathtt{d}\} = \{2,3\}$, …, $\{\mathtt{g},\mathtt{h}\} = \{6,7\}$, $\{\mathtt{a},\mathtt{c},\mathtt{e},\mathtt{g}\} = \{\mathtt{A},\mathtt{C},\mathtt{E},\mathtt{G}\}$, $\{\mathtt{b},\mathtt{d},\mathtt{f},\mathtt{h}\} = \{\mathtt{B},\mathtt{D},\mathtt{F},\mathtt{H}\}$, $\{\mathtt{A},\mathtt{B}\} = \{3,2\}$, $\{\mathtt{C},\mathtt{D}\} = \{4,1\}$, …, $\{\mathtt{G},\mathtt{H}\} = \{5,7\}$. Starting at the bottom, let us choose $\mathtt{h} = 7$, because we don't wish to disturb the contents of that line unless necessary. Then the following choices are *forced*:

$$\mathtt{H}=7;\ \mathtt{G}=5;\ \mathtt{e}=5;\ \mathtt{f}=4;\ \mathtt{D}=4;\ \mathtt{C}=1;\ \mathtt{a}=1;\ \mathtt{b}=0;\ \mathtt{F}=0;\ \mathtt{E}=6;\ \mathtt{g}=6. \quad (73)$$

If we had chosen $\mathtt{h} = 6$, the forcing pattern would have been similar but reversed,

$$\mathtt{F}=6;\ \mathtt{E}=0;\ \mathtt{a}=0;\ \mathtt{b}=1;\ \mathtt{D}=1;\ \mathtt{C}=4;\ \mathtt{e}=4;\ \mathtt{f}=5;\ \mathtt{H}=5;\ \mathtt{G}=7;\ \mathtt{g}=7. \quad (74)$$

Options (73) and (74) can both be completed by choosing either $\mathtt{d} = 3$ (hence $\mathtt{B} = 3$, $\mathtt{A} = 2$, $\mathtt{c} = 2$) or $\mathtt{d} = 2$ (hence $\mathtt{B} = 2$, $\mathtt{A} = 3$, $\mathtt{c} = 3$).

In general the forcing pattern will go in cycles, no matter what permutation we begin with. To see this, consider the graph on eight vertices {`ab`, `cd`, `ef`, `gh`, `AB`, `CD`, `EF`, `GH`} that has an edge from `uv` to `UV` whenever the pair of inputs connected to `uv` has an element in common with the pair of outputs connected to `UV`. Thus, in our example the edges are `ab` — `EF`, `ab` — `CD`, `cd` — `AB`, `cd` — `AB`, `ef` — `CD`, `ef` — `GH`, `gh` — `EF`, `gh` — `GH`. We have a “double bond” between `cd` and `AB`, since the inputs connected to `c` and `d` are exactly the outputs connected to `A` and `B`; subject to this slight bending of the strict definition of a graph, we see that each vertex is adjacent to exactly two other vertices, and lowercase vertices are always adjacent to uppercase ones. Therefore the graph

cycles in a graph
transpose
matrix transposition

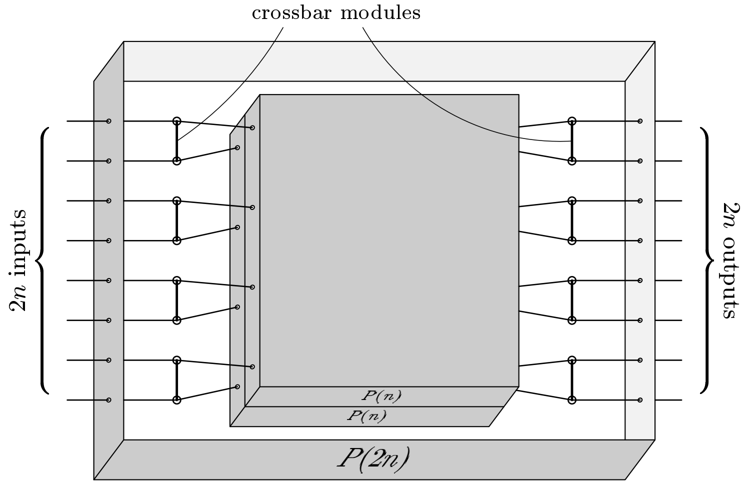


Fig. 12. The inside of a black box $P(2n)$ that permutes $2n$ elements in all possible ways, when $n > 1$. (Illustrated for $n = 4$.)

always consists of disjoint cycles of even length. In our example, the cycles are

$$\texttt{ab} \begin{smallmatrix} \diagup\ \texttt{EF} - \texttt{gh}\ \diagdown \\ \diagdown\ \texttt{CD} - \texttt{ef}\ \diagup \end{smallmatrix} \texttt{GH} \qquad \texttt{cd} = \texttt{AB}, \tag{75}$$

where the longer cycle corresponds to (73) and (74). If there are k different cycles, there will be 2^k different ways to specify the behavior of the first and last columns of crossbars.

To complete the network, we can process the inner 4-element permutations in the same way; and *any* 2^d-element permutation is achievable in this same recursive fashion. The resulting crossbar settings determine the masks θ_j and $\hat{\theta}_j$ of (71). Some choices of crossbars may lead to a mask that is entirely zero; then we can eliminate the corresponding stage of the computation.

If the input and output are identical on the bottom lines of the network, our construction shows how to ensure that none of the crossbars touching those lines are active. For example, the 64-bit algorithm in (71) could be used also with a 60-bit register, without needing the four extra bits for any intermediate results.

Of course we can often beat the general procedure of (71) in special cases. For example, exercise 52 shows that method (71) needs nine swapping steps to transpose an 8×8 matrix, but in fact three swaps suffice:

Given	7-swap	14-swap	28-swap
00 01 02 03 04 05 06 07	00 **10** 02 **12** 04 **14** 06 **16**	00 10 **20 30** 04 14 **24 34**	00 10 20 30 **40 50 60 70**
10 11 12 13 14 15 16 17	**01** 11 **03** 13 **05** 15 **07** 17	01 11 **21 31** 05 15 **25 35**	01 11 21 31 **41 51 61 71**
20 21 22 23 24 25 26 27	20 **30** 22 **32** 24 **34** 26 **36**	**02 12** 22 32 **06 16** 26 36	02 12 22 32 **42 52 62 72**
30 31 32 33 34 35 36 37	**21** 31 **23** 33 **25** 35 **27** 37	**03 13** 23 33 **07 17** 27 37	03 13 23 33 **43 53 63 73**
40 41 42 43 44 45 46 47	40 **50** 42 **52** 44 **54** 46 **56**	40 50 **60 70** 44 54 **64 74**	**04 14 24 34** 44 54 64 74
50 51 52 53 54 55 56 57	**41** 51 **43** 53 **45** 55 **47** 57	41 51 **61 71** 45 55 **65 75**	**05 15 25 35** 45 55 65 75
60 61 62 63 64 65 66 67	60 **70** 62 **72** 64 **74** 66 **76**	**42 52** 62 72 **46 56** 66 76	**06 16 26 36** 46 56 66 76
70 71 72 73 74 75 76 77	**61** 71 **63** 73 **65** 75 **67** 77	**43 53** 63 73 **47 57** 67 77	**07 17 27 37** 47 57 67 77

The "perfect shuffle" is another bit permutation that arises frequently in practice. If $x = (\ldots x_2x_1x_0)_2$ and $y = (\ldots y_2y_1y_0)_2$ are any 2-adic integers, we define $x \ddagger y$ ("x zip y," the *zipper function* of x and y) by interleaving their bits:

$$x \ddagger y \;=\; (\ldots x_2y_2x_1y_1x_0y_0)_2. \tag{76}$$

This operation has important applications to the representation of 2-dimensional data, because a small change in either x or y usually causes only a small change in $x \ddagger y$. Notice also that the magic mask constants (47) satisfy

$$\mu_k \ddagger \mu_k \;=\; \mu_{k+1}. \tag{77}$$

If x appears in the left half of a register and y appears in the right half, a perfect shuffle is the permutation that changes the register contents to $x \ddagger y$.

A sequence of $d-1$ swapping steps will perfectly shuffle a 2^d-bit register; in fact, exercise 53 shows that there are several ways to achieve this. Once again, therefore, we are able to improve on the $(2d-1)$-step method of (71) and Fig. 12.

Conversely, suppose we're given the shuffled value $z = x \ddagger y$ in a 2^d-bit register; is there an efficient way to extract the original value of y? Sure: If the $d-1$ swaps that do a perfect shuffle are performed in reverse order, they'll undo the shuffle and recover both x and y. But if only y is wanted, we can save half of the work: Start with $y \leftarrow z \,\&\, \mu_0$; then set $y \leftarrow (y \oplus (y \gg 2^{k-1})) \,\&\, \mu_k$ for $k = 1$, $\ldots$, $d-1$. For example, when $d = 3$ this procedure goes $(0y_30y_20y_10y_0)_2 \mapsto (00y_3y_200y_1y_0)_2 \mapsto (0000y_3y_2y_1y_0)_2$. "Divide and conquer" conquers again.

Consider now a more general problem, where we want to extract and compress an *arbitrary* subset of a register's bits. Suppose we're given a 2^d-bit word $z = (z_{2^d-1}\ldots z_1z_0)_2$ and a mask $\chi = (\chi_{2^d-1}\ldots\chi_1\chi_0)_2$ that has s 1-bits; thus $\nu\chi = s$. The problem is to assemble the compact subword

$$y \;=\; (y_{s-1}\ldots y_1y_0)_2 \;=\; (z_{j_{s-1}}\ldots z_{j_1}z_{j_0})_2, \tag{78}$$

where $j_{s-1} > \cdots > j_1 > j_0$ are the indices where $\chi_j = 1$. For example, if $d = 3$ and $\chi = (10110010)_2$, we want to transform $z = (y_3x_3y_2y_1x_2x_1y_0x_0)_2$ into $y = (y_3y_2y_1y_0)_2$. (The problem of going from $x \ddagger y$ to y, considered above, is the special case $\chi = \mu_0$.) We know from (71) that y can be found by δ-swapping, at most $2d-1$ times; but in this problem the relevant data always moves to the right, so we can speed things up by doing *shifts* instead of swaps.

Let's say that a δ-shift of x with mask θ is the operation

$$x \;\leftarrow\; x \oplus \bigl((x \oplus (x \gg \delta)) \,\&\, \theta\bigr), \tag{79}$$

which changes bit x_j to $x_{j+\delta}$ if θ has 1 in position j, otherwise it leaves x_j unchanged. Guy Steele discovered that there always exist masks $\theta_0, \theta_1, \ldots\ \theta_{d-1}$ so that the general extraction problem (78) can be solved with a few δ-shifts:

$$\begin{aligned}&\text{Start with } x \leftarrow z\text{; then do a } 2^k\text{-shift of } x \text{ with mask } \theta_k,\\ &\quad\text{for } k = 0, 1, \ldots, d-1\text{; finally set } y \leftarrow x.\end{aligned} \tag{80}$$

In fact, the idea for finding appropriate masks is surprisingly simple. Suppose exactly t_l of the bits need to travel l places to the right. When we do a 2^k-shift, we'll use θ_k to select those bits for which $\lfloor l/2^k \rfloor$ is odd.

perfect shuffle
2-adic integers
interleaving, see zipper function, perf shuffle
2-dimensional data
magic mask
Divide and conquer
extract and compress
mask
packing
δ-shift
Steele

For example, when $d = 3$ and $\chi = (10110010)_2$ we have $(t_0, t_1, t_2, t_3, t_4) = (0, 1, 0, 2, 5)$, including 0s to be shifted in from the left so that $\sum_l t_l = 2^d$. If we set $\theta_0 = (00111011)_2$, $\theta_1 = (00011110)_2$, $\theta_2 = (11111000)_2$, then (80) maps

$$(y_3x_3y_2y_1x_2x_1y_0x_0)_2 \mapsto (y_3x_3x_3y_2y_1x_1x_1y_0)_2 \mapsto (y_3x_3x_3x_3x_3y_2y_1y_0)_2 \mapsto (0000y_3y_2y_1y_0)_2.$$

Exercise 69 proves that the bits being extracted will never interfere with each other during their journey. Furthermore, there's a slick way to compute the necessary masks θ_k dynamically in $O(d^2)$ steps (see exercise 70).

A "sheep-and-goats" operation has been suggested for computer hardware, extending (78) to produce the general unshuffled word

$$z \,\cdot\!|\!\cdot\, \chi = (x_{r-1} \ldots x_1x_0y_{s-1} \ldots y_1y_0)_2 = (z_{i_{r-1}} \ldots z_{i_1}z_{i_0}z_{j_{s-1}} \ldots z_{j_1}z_{j_0})_2; \qquad (81)$$

here $i_{r-1} > \cdots > i_1 > i_0$ are the indices where $\chi_i = 0$. Any permutation of 2^d bits is achievable via at most d sheep-and-goats operations (see exercise 73).

Shifting also allows us to go beyond permutations, to arbitrary *mappings* of bits within a register. Suppose we want to transform

$$x = (x_{2^d-1} \ldots x_1x_0)_2 \quad \mapsto \quad x^\varphi = (x_{(2^d-1)\varphi} \ldots x_{1\varphi}x_{0\varphi})_2, \qquad (82)$$

where φ is any of the $(2^d)^{2^d}$ functions from the set $\{0, 1, \ldots, 2^d - 1\}$ into itself. K. M. Chung and C. K. Wong [*IEEE Transactions* **C-29** (1980), 1029–1032] discovered an attractive way to do this in $O(d)$ steps by using *cyclic* δ-shifts, which are like (79) except that we set

$$x \;\leftarrow\; x \oplus \bigl((x \oplus (x \gg \delta) \oplus (x \ll (2^d - \delta)))\,\&\,\theta\bigr). \qquad (83)$$

Their idea is to let c_l be the number of indices j such that $j\varphi = l$, for $0 \le l < 2^d$. Then they find masks θ_0, θ_1, …, θ_{d-1} with the property that a cyclic 2^k-shift of x with mask θ_k, done successively for $0 \le k < d$, will transform x into a number x' that contains exactly c_l copies of bit x_l for each l. Finally the general permutation procedure (71) can be used to change $x' \mapsto x^\varphi$.

For example, suppose $d = 3$ and $x^\varphi = (x_3x_1x_1x_0x_3x_7x_5x_5)_2$. Then we have $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7) = (1, 2, 0, 2, 0, 2, 0, 1)$. Using masks $\theta_0 = (00011100)_2$, $\theta_1 = (01001001)_2$, and $\theta_2 = (00100000)_2$, three cyclic 2^k-shifts now take $x = (x_7x_6x_5x_4x_3x_2x_1x_0)_2 \mapsto (x_7x_6x_5x_5x_4x_3x_1x_0)_2 \mapsto (x_7x_0x_5x_5x_5x_3x_1x_3)_2 \mapsto (x_7x_0x_1x_5x_5x_3x_1x_3)_2 = x'$. Then, five δ-swaps: $x' \mapsto (x_0x_7x_5x_1x_3x_5x_3x_1)_2 \mapsto (x_0x_7x_5x_1x_3x_1x_3x_5)_2 \mapsto (x_0x_1x_3x_1x_3x_7x_5x_5)_2 \mapsto (x_3x_1x_0x_1x_3x_7x_5x_5)_2 \mapsto (x_3x_1x_1x_0x_3x_7x_5x_5)_2 = x^\varphi$; we're done! Of course any 8-bit mapping can be achieved more quickly by brute force, one bit at a time; the method of Chung and Wong becomes much more impressive in a 256-bit register. Even with MMIX's 64-bit registers it's pretty good, needing at most 96 cycles in the worst case.

To find θ_0, we use the fact that $\sum c_l = 2^d$, and we look at $\Sigma_{\rm even} = \sum c_{2l}$ and $\Sigma_{\rm odd} = \sum c_{2l+1}$. If $\Sigma_{\rm even} = \Sigma_{\rm odd} = 2^{d-1}$, we can set $\theta_0 = 0$ and omit the cyclic 1-shift. But if, say, $\Sigma_{\rm even} < \Sigma_{\rm odd}$, we find an even l with $c_l = 0$. Cyclically shifting the bits $l, l+1, \ldots, l+t$ (modulo 2^d) for some t will produce new counts $(c'_0, \ldots, c'_{2^d-1})$ for which $\Sigma'_{\rm even} = \Sigma'_{\rm odd} = 2^{d-1}$; so $\theta_0 = 2^l + \cdots + 2^{(l+t) \bmod 2^d}$. Then we can deal with the bits in even and odd positions separately, using the same method, until getting down to 1-bit subwords. Exercise 74 has the details.

sheep-and-goats
notation $z \,\cdot\!|\!\cdot\, \chi$
mappings
Chung
Wong
cyclic
masks
pi, as "random" example
recursively

Working with fragmented fields. Instead of extracting bits from various parts of a word and gathering them together, we can often manipulate those bits directly in their original positions.

For example, suppose we want to run through all subsets of a given set U, where (as usual) the set is specified by a mask χ such that $[k \in U] = (\chi \gg k)\,\&\,1$. If $x \subseteq \chi$ and $x \neq \chi$, there's an easy way to calculate the next largest subset of U in lexicographic order, namely the smallest integer $x' > x$ such that $x' \subseteq \chi$:

$$x' \;=\; (x - \chi)\,\&\,\chi. \tag{84}$$

In the special case when $x = 0$ and $\chi \neq 0$, we've already seen in (37) that this formula produces the rightmost bit of χ, which corresponds to the lexicographically smallest nonempty subset of U.

Why does formula (84) work? Imagine adding 1 to the number $x \mid \bar{\chi}$, which has 1s wherever χ is 0. A carry will propagate through those 1s until it reaches the rightmost bit position where x has a 0 and χ has a 1; furthermore all bits to the right of that position will become zero. Therefore $x' = ((x \mid \bar{\chi}) + 1)\,\&\,\chi$. But we have $(x \mid \bar{\chi}) + 1 = (x + \bar{\chi}) + 1 = x + (\bar{\chi} + 1) = x - \chi$ when $x \subseteq \chi$. QED.

Notice further that $x' = 0$ if and only if $x = \chi$. So we'll know when we've found the largest subset. Exercise 79 shows how to go back to x, given x'.

We might also want to run through all elements of a *subcube*— for example, to find all bit patterns that match a specification like $*10*1*01$, consisting of 0s, 1s, and $*$s (don't-cares). Such a specification can be represented by asterisk codes $a = (a_{n-1} \ldots a_0)_2$ and bit codes $b = (b_{n-1} \ldots b_0)_2$, as in exercise 7.1.1–30; our example corresponds to $a = (10010100)_2$, $b = (01001001)_2$. The problem of enumerating all subsets of a set is the special case where $a = \chi$ and $b = 0$. In the more general subcube problem, the successor of a given bit pattern x is

$$x' \;=\; ((x - (a + b))\,\&\,a) + b. \tag{85}$$

Suppose the bits of $z = (z_{n-1} \ldots z_0)_2$ have been stitched together from two subwords $x = (x_{r-1} \ldots x_0)_2$ and $y = (y_{s-1} \ldots y_0)_2$, where $r + s = n$, using an arbitrary mask χ for which $\nu\chi = s$ to govern the stitching. For example, $z = (y_2 x_4 x_3 y_1 x_2 y_0 x_1 x_0)_2$ when $n = 8$ and $\chi = (10010100)_2$. We can think of z as a "scattered accumulator," in which alien bits x_i lurk among friendly bits y_j. From this viewpoint the problem of finding successive elements of a subcube is essentially the problem of computing $y + 1$ inside a scattered accumulator z, without changing the value of x. The sheep-and-goats operation (81) would untangle x and y; but it's expensive, and (85) shows that we can solve the problem without it. We can, in fact, compute $y + y'$ when $y' = (y'_{s-1} \ldots y'_0)_2$ is *any* value inside a scattered accumulator z', if y and y' both appear in the positions specified by χ: Consider $t = z\,\&\,\chi$ and $t' = z'\,\&\,\chi$. If we form the sum $(t \mid \bar{\chi}) + t'$, all carries that occur in a normal addition $y + y'$ will propagate through the blocks of 1s in $\bar{\chi}$, just as if the scattered bits were adjacent. Thus

$$((z\,\&\,\chi) + (z'\,\&\,\chi) + \bar{\chi})\,\&\,\chi \tag{86}$$

is the sum of y and y', modulo 2^s, scattered according to the mask χ.

fragmented fields
subsets
mask
lexicographic order
subcube
don't-cares
asterisk codes
bit codes
scattered accumulator
sheep-and-goats
carries
scattered sum

Tweaking several bytes at once. Instead of concentrating on the data in one field within a word, we often want to deal simultaneously with two or more subwords, performing calculations on each of them in parallel. For example, many applications need to process long sequences of bytes, and we can gain speed by acting on eight bytes at a time; we might as well use all 64 bits that our machine provides. General multibyte techniques were introduced by Leslie Lamport in *CACM* **18** (1975), 471–475, and subsequently extended by many programmers.

Suppose first that we simply wish to take two sequences of bytes and find their sum, regarding them as coordinates of vectors, doing arithmetic modulo 256 in each byte. Algebraically speaking, we're given 8-byte vectors $x = (x_7 \ldots x_1 x_0)_{256}$ and $y = (y_7 \ldots y_1 y_0)_{256}$; we want to compute $z = (z_7 \ldots z_1 z_0)_{256}$, where $z_j = (x_j + y_j) \bmod 256$ for $0 \le j < 8$. Ordinary addition of x to y doesn't quite work, because we need to prevent carries from propagating between bytes. So we separate out the high-order bits and deal with them separately:

$$z \leftarrow (x \oplus y) \,\&\, h, \qquad \text{where } h = \#\texttt{8080808080808080};$$
$$z \leftarrow ((x \,\&\, \bar{h}) + (y \,\&\, \bar{h})) \oplus z. \tag{87}$$

The total time for `MMIX` to do this is $6v$, plus $3\mu + 3v$ if we also count the time to load x, load y, and store z. By contrast, eight one-byte additions (`LDBU`, `LDBU`, `ADDU`, and `STBU`, repeated eight times) would cost $8 \times (3\mu + 4v) = 24\mu + 32v$. Parallel *subtraction* of bytes is just as easy (see exercise 88).

We can also compute bytewise *averages*, with $z_j = \lfloor (x_j + y_j)/2 \rfloor$ for each j:

$$z \leftarrow ((x \oplus y) \,\&\, \bar{l}) \gg 1, \qquad \text{where } l = \#\texttt{0101010101010101};$$
$$z \leftarrow (x \,\&\, y) + z. \tag{88}$$

This elegant trick, suggested by H. G. Dietz, is based on the well-known formula

$$x + y \;=\; (x \oplus y) + ((x \,\&\, y) \ll 1) \tag{89}$$

for radix-2 addition. (We can implement (88) with four `MMIX` instructions, not five, because a single `MOR` operation will change $x \oplus y$ to $((x \oplus y) \,\&\, \bar{l}) \gg 1$.)

Exercises 88–93 and 100–104 develop these ideas further, showing how to do mixed-radix arithmetic, as well as such things as the addition and subtraction of vectors whose components are treated modulo m when m needn't be a power of 2.

In essence, we can regard the bits, bytes, or other subfields of a register as if they were elements of an array of independent microprocessors, acting independently on their own subproblems yet tightly synchronized, and communicating with each other via shift instructions and carry bits. Computer designers have been interested for many years in the development of parallel processors with a so-called SIMD architecture, namely a "Single Instruction stream with Multiple Data streams"; see, for example, S. H. Unger, *Proc. IRE* **46** (1958), 1744–1750. The increased availability of 64-bit registers has meant that programmers of ordinary sequential computers are now able to get a taste of SIMD processing. Indeed, computations such as (87), (88), and (89) are called SWAR methods— "SIMD Within A Register," a name coined by R. J. Fisher and H. G. Dietz [see *Lecture Notes in Computer Science* **1656** (1999), 290–305].

packed data, operating on++
Lamport
parallel processing of subwords
multibyte processing
multibyte addition
carries
averages
Dietz
radix-2 addition
MMIX
MOR
shift instructions
carry bits
parallel processors
SIMD architecture
Unger
SWAR
Fisher
Dietz

Of course bytes often contain alphabetic data as well as numbers, and one of the most common programming tasks is to search through a long string of characters in order to find the first appearance of some particular byte value. For example, strings are often represented as a sequence of nonzero bytes terminated by 0. In order to locate the end of a string quickly, we need a fast way to determine whether all eight bytes of a given word x are nonzero (because they usually are). Several fairly good solutions to this problem were found by Lamport and others; but Alan Mycroft discovered in 1987 that *three* instructions actually suffice:

$$t \;\leftarrow\; h \,\&\, (x-l) \,\&\, \bar{x}, \tag{90}$$

where h and l appear in (87) and (88). If each byte x_j is nonzero, t will be zero; for $(x_j-1)\&\bar{x}_j$ will be $2^{\rho x_j}-1$, which is always less than $\#\mathtt{80} = 2^7$. But if $x_j = 0$, while its right neighbors $x_{j-1}, \ldots, x_0$ (if any) are all nonzero, the subtraction $x-l$ will produce $\#\mathtt{ff}$ in byte j, and t will be nonzero. In fact, ρt will be $8j+7$.

Caution: Although the computation in (90) pinpoints the *rightmost* zero byte of x, we cannot deduce the position of the *leftmost* zero byte from the value of t alone. (See exercise 94.) In this respect the little-endian convention proves to be preferable to the corresponding big-endian behavior. An application that needs to locate the leftmost zero byte can use (90) to skip quickly over nonzeros, but then it must fall back on a slower method when the search has been narrowed down to eight finalists. The following 4-operation formula produces a completely precise test value $t = (t_7 \ldots t_1 t_0)_{256}$, in which $t_j = 128[x_j = 0]$ for each j:

$$t \;\leftarrow\; h \,\&\, {\sim}(x \mid ((x \mid h) - l)). \tag{91}$$

The leftmost nonzero byte of x is now x_j, where $\lambda t = 8j+7$.

Incidentally, the single MMIX instruction 'BDIF t,l,x' solves the zero-byte problem immediately by setting each byte t_j of t to $[x_j = 0]$, because $1 \dot{-} x = [x = 0]$. But we are primarily interested here in fairly universal techniques that don't rely on exotic hardware; MMIX's special features will be discussed later.

Now that we know a fast way to find the first 0, we can use the same ideas to search for *any* desired byte value. For example, to test if any byte of x is the newline character ($\#\mathtt{a}$), we simply look for a zero byte in $x \oplus \#\mathtt{0a0a0a0a0a0a0a0a}$.

And these techniques also open up many other doors. Suppose, for instance, that we want to compute $z = (z_7 \ldots z_1 z_0)_{256}$ from x and y, where $z_j = x_j$ when $x_j = y_j$ but $z_j =$ '*' when $x_j \neq y_j$. (Thus if $x =$ "beaching" and $y =$ "belching", we're supposed to set $z \leftarrow$ "be*ching".) It's easy:

$$\begin{aligned} &t \leftarrow h \,\&\, ((x \oplus y) \mid (((x \oplus y) \mid h) - l));\\ &m \leftarrow (t \ll 1) - (t \gg 7);\\ &z \leftarrow x \oplus ((x \oplus \texttt{"********"}) \,\&\, m). \end{aligned} \tag{92}$$

The first step uses (91) to flag the high-order bits in each byte where $x_j \neq y_j$. The next step creates a mask that highlights those bytes; the mask is $\#\mathtt{00}$ if $x_j = y_j$ and $\#\mathtt{ff}$ otherwise. And the last step, which could also be written $z \leftarrow (x \,\&\, \overline{m}) \mid (\texttt{"********"} \,\&\, m)$, sets $z_j \leftarrow x_j$ or $z_j \leftarrow$ '*', depending on the mask.

alphabetic data
strings
Lamport
Mycroft
ruler function
little-endian convention
big-endian
dotminus
equality of bytes
newline
flag: A 1-bit indicator
mask

Operations (90) and (91) were originally designed as tests for bytes that are zero; but a closer look reveals that we can more wisely regard them as tests for bytes that are less than 1. Indeed, if we replace l by $c \cdot l = (cccccccc)_{256}$ in either formula, where c is any positive constant ≤ 128, we can use (90) or (91) to see if x contains any bytes that are less than c. Furthermore the comparison values c need not be the same in every byte position; and with a bit more work we can also do bytewise comparison in the cases where $c > 128$. Here's an 8-step formula that sets $t_j \leftarrow 128[x_j < y_j]$ for each byte position j in the test word t:

$$t \leftarrow h \mathbin{\&} {\sim}\langle x\bar{y}z\rangle, \qquad \text{where } z = (x \mid h) - (y \mathbin{\&} \bar{h}). \tag{93}$$

(See exercise 96.) The median operation in this general formula can often be simplified; for example, (93) reduces to (91) when $y = l$, because $\langle x1z\rangle = x \mid z$.

Once we've found a nonzero t in (90) or (91) or (93), we might want to compute ρt or λt in order to discover the index j of the rightmost or leftmost byte that has been flagged. The problem of calculating ρ or λ is now simpler than before, since t can take on only 256 different values. Indeed, the operation

$$j \leftarrow \mathit{table}[((a \cdot t) \bmod 2^{64}) \gg 56], \quad \text{where } a = \frac{2^{56}-1}{2^7-1}, \tag{94}$$

now suffices to compute j, given an appropriate 256-byte table. And the multiplication here can often be performed faster by doing three shift-and-add operations, "$t \leftarrow t + (t \ll 7)$, $t \leftarrow t + (t \ll 14)$, $t \leftarrow t + (t \ll 28)$," instead.

Broadword computing. We've now seen more than a dozen ways in which a computer's bitwise operations can produce astonishing results at high speed, and the exercises below contain many more such surprises.

Elwyn Berlekamp has remarked that computer chips containing N flip-flops continue to be built with ever larger values of N, yet in practice only $O(\log N)$ of those components are flipping or flopping at any given moment. The surprising effectiveness of bitwise operations suggests that computers of the future might make use of this untapped potential by having enhanced memory units that are able to do efficient n-bit computations for fairly large values of n. To prepare for that day, we ought to have a good name for the concept of manipulating "wide words." Lyle Ramshaw has suggested the pleasant term *broadword*, so that we can speak of n-bit quantities as broadwords of width n.

Many of the methods we've discussed are *2-adic*, in the sense that they work correctly with binary numbers that have arbitrary (even infinite) precision. For example, the operation $x \mathbin{\&} -x$ always extracts $2^{\rho x}$, the least significant 1 bit of any nonzero 2-adic integer x. But other methods have an inherently broadword nature, such as the methods that use $O(d)$ steps to perform sideways addition or bit permutation of 2^d-bit words. Broadword computing is the art of dealing with n-bit words, when n is a parameter that is not extremely small.

Some broadword algorithms are of theoretical interest only, because they are efficient only in an asymptotic sense when n exceeds the size of the universe. But others are eminently practical even when $n = 64$. And in general, a broadword mindset often suggests good techniques.

comparison of bytes
bytes, testing relative order of
median operation
ρ
λ
Berlekamp
Ramshaw
2-adic

Fredman
Willard
Brodal
Paterson
Knuth, DE
pattern
storage allocation

One fascinating-but-impractical fact about broadword operations is the discovery by M. L. Fredman and D. E. Willard that $O(1)$ broadword steps suffice to evaluate the function $\lambda x = \lfloor \lg x \rfloor$ for any nonzero n-bit number x, no matter how big n is. Here is their remarkable scheme, when $n = g^2$ and g is a power of 2:

$$\begin{aligned}
&t_1 \leftarrow h \mathbin{\&} (x \mid ((x \mid h) - l)), \quad \text{where } h = 2^{g-1}l \text{ and } l = (2^n-1)/(2^g-1);\\
&y \leftarrow (((a \cdot t_1) \bmod 2^n) \gg (n-g)) \cdot l, \quad \text{where } a = (2^{n-g}-1)/(2^{g-1}-1);\\
&t_2 \leftarrow h \mathbin{\&} (y \mid ((y \mid h) - b)), \quad \text{where } b = (2^{n+g}-1)/(2^{g+1}-1);\\
&m \leftarrow (t_2 \ll 1) - (t_2 \gg (g-1)),\ m \leftarrow m \oplus (m \gg g);\\
&z \leftarrow (((l \cdot (x \mathbin{\&} m)) \bmod 2^n) \gg (n-g)) \cdot l;\\
&t_3 \leftarrow h \mathbin{\&} (z \mid ((z \mid h) - b));\\
&\lambda \leftarrow ((l \cdot ((t_2 \gg (2g - \lg g - 1)) + (t_3 \gg (2g-1)))) \bmod 2^n) \gg (n-g).
\end{aligned} \tag{95}$$

(See exercise 106.) The method fails to be practical because five of these 29 steps are multiplications, so they aren't really "bitwise" operations. In fact, we'll prove later that multiplication by a constant requires at least $\Omega(\log n)$ bitwise steps.

A multiplication-free way to find λx, with only $O(\log\log n)$ bitwise broadword operations, was discovered in 1997 by Gerth Brodal, whose method is even more remarkable than (95). It is based on a formula analogous to (49),

$$\lambda x \;=\; [\lambda x = \lambda(x \mathbin{\&} \bar\mu_0)] + 2[\lambda x = \lambda(x \mathbin{\&} \bar\mu_1)] + 4[\lambda x = \lambda(x \mathbin{\&} \bar\mu_2)] + \cdots, \tag{96}$$

and the fact that the relation $\lambda x = \lambda y$ is easily tested (see (58)):

Algorithm B (*Binary logarithm*). This algorithm uses n-bit operations to compute $\lambda x = \lfloor \lg x \rfloor$, assuming that $0 < x < 2^n$ and $n = d \cdot 2^d$.

B1. [Scale down.] Set $\lambda \leftarrow 0$. Then set $\lambda \leftarrow \lambda + 2^k$ and $x \leftarrow x \gg 2^k$ if $x \ge 2^{2^k}$, for $\lceil \lg n \rceil > k \ge d$.

B2. [Replicate.] (At this point $0 < x < 2^{2^d}$; the remaining task is to increase λ by $\lfloor \lg x \rfloor$. We will replace x by d copies of itself, in 2^d-bit fields.) Set $x \leftarrow x \mid (x \ll 2^{d+k})$ for $0 \le k < \lceil \lg d \rceil$.

B3. [Change leading bits.] Set $y \leftarrow x \mathbin{\&} {\sim}(\mu_{d,d-1} \ldots \mu_{d,1}\mu_{d,0})_{2^{2^d}}$. (See (48).)

B4. [Compare all fields.] Set $t \leftarrow h \mathbin{\&} (y \mid ((y \mid h) - (x \oplus y)))$, where $h = (2^{2^d-1} \ldots 2^{2^d-1} 2^{2^d-1})_{2^{2^d}}$.

B5. [Compress bits.] Set $t \leftarrow (t + (t \ll (2^{d+k} - 2^k))) \bmod 2^n$ for $0 \le k < \lceil \lg d \rceil$.

B6. [Finish.] Finally, set $\lambda \leftarrow \lambda + (t \gg (n-d))$. ▮

This algorithm is actually competitive with (56) when $n = 64$ (see exercise 107).

Another surprisingly efficient broadword algorithm was discovered in 2006 by M. S. Paterson and the author, who considered the problem of identifying all occurrences of the pattern 01^r in a given n-bit binary string. This problem, which is related to storage allocation, is equivalent to computing

$$q \;=\; \bar x \mathbin{\&} (x \ll 1) \mathbin{\&} (x \ll 2) \mathbin{\&} (x \ll 3) \mathbin{\&} \cdots \mathbin{\&} (x \ll r) \tag{97}$$

2-adic chain++++
broadword chain++++
branchless+++
table lookup by shifting

when $x = (x_{n-1} \ldots x_1 x_0)_2$ is given. For example, when $n = 16$, $r = 3$, and $x = (1110111101100111)_2$, we have $q = (0001000000001000)_2$. One might expect intuitively that $\Omega(\log r)$ bitwise operations would be needed. But in fact the following 21-step computation does the job for all $n > r > 0$: Let $s = \lceil r/2 \rceil$, $l = \sum_{k \ge 0} 2^{ks} \bmod 2^n$, $h = (2^{s-1} l) \bmod 2^n$, and $a = \bigl(\sum_{k\ge 0}(-1)^{k+1} 2^{2ks}\bigr) \bmod 2^n$.

$$\begin{aligned} &y \leftarrow h \,\&\, x \,\&\, ((x \,\&\, \bar{h}) + l);\\ &t \leftarrow (x + y) \,\&\, \bar{x} \,\&\, -2^r;\\ &u \leftarrow t \,\&\, a,\;\; v \leftarrow t \,\&\, (a \ll 2s);\\ &m \leftarrow (u - (u \gg r)) \mid (v - (v \gg r));\\ &q \leftarrow t \,\&\, ((x \,\&\, m) + ((t \gg r) \,\&\, {\sim}(m \ll 1))). \end{aligned} \tag{98}$$

Exercise 111 explains why these machinations are valid. The method has little or no practical value; there's an easy way to evaluate (97) in $2\lceil \lg r \rceil + 2$ steps, so (98) is not advantageous until $r > 512$. But (98) is another indication of the unexpected power of broadword methods.

***Lower bounds.** Indeed, the existence of so many tricks and techniques makes it natural to wonder whether we've only been scratching the surface. Are there many more incredibly fast methods, still waiting to be discovered? A few theoretical results are known by which we can derive certain limitations on what is possible, although such studies are still in their infancy.

Let's say that a *2-adic chain* is a sequence $(x_0, x_1, \ldots, x_r)$ of 2-adic integers in which each element x_i for $i > 0$ is obtained from its predecessors via bitwise manipulation. More precisely, we want the steps of the chain to be defined by binary operations

$$x_i \;=\; x_{j(i)} \circ_i x_{k(i)} \quad\text{or}\quad c_i \circ_i x_{k(i)} \quad\text{or}\quad x_{j(i)} \circ_i c_i, \tag{99}$$

where each $\circ_i$ is one of the operators $\{+, -, \&, |, \oplus, \equiv, \subset, \supset, \overline{\subset}, \overline{\supset}, \overline{\wedge}, \overline{\vee}, \ll, \gg\}$ and each c_i is a constant. Furthermore, when the operator $\circ_i$ is a left shift or right shift, the amount of shift must be a positive integer constant; operations such as $x_{j(i)} {\ll} x_{k(i)}$ or $c_i {\gg} x_{k(i)}$ are *not* permitted. (Without the latter restriction we couldn't derive meaningful lower bounds, because *every* 0–1 valued function of a nonnegative integer x would be computable in two steps as "$(c \gg x) \,\&\, 1$" for some constant c.)

Similarly, a *broadword chain* of width n, also called an n-bit broadword chain, is a sequence $(x_0, x_1, \ldots, x_r)$ of n-bit numbers subject to essentially the same restrictions, where n is a parameter and all operations are performed modulo 2^n. Broadword chains behave like 2-adic chains in many ways, but subtle differences can arise because of the information loss that occurs at the left of n-bit computations (see exercise 113).

Both types of chains compute a function $f(x) = x_r$ when we start them out with a given value $x = x_0$. Exercise 114 shows that an mn-bit broadword chain is able to do m essentially simultaneous evaluations of any function that is computable with an n-bit chain. Our goal is to study the *shortest* chains that are able to evaluate a given function f.

shift sets
division, by 10
monus

Any 2-adic or broadword chain $(x_0, x_1, \ldots, x_r)$ has a sequence of "shift sets" $(S_0, S_1, \ldots, S_r)$ and "bounds" $(B_0, B_1, \ldots, B_r)$, defined as follows: Start with $S_0 = \{0\}$ and $B_0 = 1$; then for $i \ge 1$, let

$$S_i = \begin{cases} S_{j(i)} \cup S_{k(i)}, \\ S_{k(i)}, \\ S_{j(i)}, \\ S_{j(i)} + c_i, \\ S_{j(i)} - c_i, \end{cases} \quad \text{and} \quad B_i = \begin{cases} M_i B_{j(i)} B_{k(i)}, & \text{if } x_i = x_{j(i)} \circ_i x_{k(i)}, \\ M_i B_{k(i)}, & \text{if } x_i = c_i \circ_i x_{k(i)}, \\ M_i B_{j(i)}, & \text{if } x_i = x_{j(i)} \circ_i c_i, \\ B_{j(i)}, & \text{if } x_i = x_{j(i)} \gg c_i, \\ B_{j(i)}, & \text{if } x_i = x_{j(i)} \ll c_i, \end{cases} \qquad (100)$$

where $M_i = 2$ if $\circ_i \in \{+, -\}$ and $M_i = 1$ otherwise, and these formulas assume that $\circ_i \notin \{\ll, \gg\}$. For example, consider the following 7-step chain:

x_i	S_i	B_i
$x_0 = x$	$\{0\}$	1
$x_1 = x_0 \mathbin{\&} -2$	$\{0\}$	1
$x_2 = x_1 + 2$	$\{0\}$	2
$x_3 = x_2 \gg 1$	$\{1\}$	2
$x_4 = x_2 + x_3$	$\{0, 1\}$	8
$x_5 = x_4 \gg 4$	$\{4, 5\}$	8
$x_6 = x_4 + x_5$	$\{0, 1, 4, 5\}$	128
$x_7 = x_6 \gg 4$	$\{4, 5, 8, 9\}$	128

(101)

(We encountered this chain in exercise 4.4–9, which proved that these operations will yield $x_7 = \lfloor x/10 \rfloor$ for $0 \le x < 160$ when performed with 8-bit arithmetic.)

To begin a theory of lower bounds, let's notice first that the high-order bits of $x = x_0$ cannot influence any low-order bits unless we shift them to the right.

Lemma A. *Given a 2-adic or broadword chain, let the binary representation of x_i be $(\ldots x_{i2} x_{i1} x_{i0})_2$. Then bit x_{ip} can depend on bit x_{0q} only if $q \le p + \max S_i$.*

Proof. By induction on i we can in fact show that, if $B_i = 1$, bit x_{ip} can depend on bit x_{0q} only if $q - p \in S_i$. Addition and subtraction, which force $B_i > 1$, allow any particular bit of their operands to affect all bits that lie to the left in the sum or difference, but not those that lie to the right. ▍

Corollary I. *The function $x \dot{-} 1$ cannot be computed by a 2-adic chain, nor can any function for which at least one bit of $f(x)$ depends on an unbounded number of bits of x.* ▍

Corollary W. *An n-bit function $f(x)$ can be computed by an n-bit broadword chain without shifts if and only if $x \equiv y$ (modulo 2^p) implies $f(x) \equiv f(y)$ (modulo 2^p) for $0 \le p < n$.*

Proof. If there are no shifts we have $S_i = \{0\}$ for all i. Thus bit x_{rp} cannot depend on bit x_{0q} unless $q \le p$. In other words we must have $x_r \equiv y_r$ (modulo 2^p) whenever $x_0 \equiv y_0$ (modulo 2^p).

Conversely, all such functions are achievable by a sufficiently long chain. Exercise 119 gives shift-free n-bit chains for the functions

$$f_{py}(x) \;=\; 2^p [x \bmod 2^{p+1} = y], \qquad \text{when } 0 \le p < n \text{ and } 0 \le y < 2^{p+1}, \quad (102)$$

from which all the relevant functions arise by addition. [H. S. Warren, Jr., generalized this result to functions of m variables in *CACM* **20** (1977), 439–441.] ▌

Warren
carry
ruler function
binary logarithm
reversal
bit permutation

Shift sets S_i and bounds B_i are important chiefly because of a fundamental lemma that is our principal tool for proving lower bounds:

Lemma B. *Let $X_{pqr} = \{x_r \,\&\, \lfloor 2^p - 2^q \rfloor \mid x_0 \in V_{pqr}\}$ in an n-bit broadword chain, where*

$$V_{pqr} \;=\; \{x \mid x \,\&\, \lfloor 2^{p+s} - 2^{q+s} \rfloor = 0 \text{ for all } s \in S_r\} \qquad (103)$$

and $p > q$. Then $|X_{pqr}| \le B_r$. (Here p and q are integers, possibly negative.)

This lemma states that at most B_r different bit patterns $x_{r(p-1)} \ldots x_{rq}$ can occur within $f(x)$, when certain intervals of bits in x are constrained to be zero.

Proof. The result certainly holds when $r = 0$. Otherwise if, for example, $x_r = x_j + x_k$, we know by induction that $|X_{pqj}| \le B_j$ and $|X_{pqk}| \le B_k$. Furthermore $V_{pqr} = V_{pqj} \cap V_{pqk}$, since $S_r = S_j \cup S_k$. Thus at most $B_j B_k$ possibilities for $(x_j + x_k) \,\&\, \lfloor 2^p - 2^q \rfloor$ arise when there's no carry into position q, and at most $B_j B_k$ when there is a carry, making a grand total of at most $B_r = 2B_j B_k$ possibilities altogether. Exercise 122 considers the other cases. ▌

We now can prove that the ruler function needs $\Omega(\log\log n)$ steps.

Theorem R. *If $n = d \cdot 2^d$, every n-bit broadword chain that computes ρx for $0 < x < 2^n$ has more than $\lg d$ steps that are not shifts.*

Proof. If there are l nonshift steps, we have $|S_r| \le 2^l$ and $B_r \le 2^{2^l - 1}$. Apply Lemma B with $p = d$ and $q = 0$, and suppose $|X_{d0r}| = 2^d - t$. Then there are t values of $k < 2^d$ such that

$$\{2^k,\, 2^{k+2^d},\, 2^{k+2\cdot 2^d},\, \ldots,\, 2^{k+(d-1)2^d}\} \;\notin\; V_{d0r}.$$

But V_{d0r} excludes at most $2^l d$ of the n possible powers of 2; so $t \le 2^l$.

If $l \le \lg d$, Lemma B tells us that $2^d - t \le B_r \le 2^{d-1}$; hence $2^{d-1} \le t \le 2^l \le d$. But this is impossible unless $d \le 2$, when the theorem clearly holds. ▌

The same proof works also for the binary logarithm function:

Corollary L. *If $n = d \cdot 2^d > 2$, every n-bit broadword chain that computes λx for $0 < x < 2^n$ has more than $\lg d$ steps that are not shifts.* ▌

By using Lemma B with $q > 0$ we can derive the stronger lower bound $\Omega(\log n)$ for bit reversal, and hence for bit permutation in general.

Theorem P. *If $2 \le g \le n$, every n-bit broadword chain that computes the g-bit reversal x^R for $0 \le x < 2^g$ has at least $\lfloor \frac{1}{3} \lg g \rfloor$ steps that are not shifts.*

Proof. Assume as above that there are l nonshifts. Let $h = \lfloor \sqrt[3]{g} \rfloor$ and suppose that $l < \lfloor \lg(h+1) \rfloor$. Then S_r is a set of at most $2^l \le \frac{1}{2}(h+1)$ shift amounts s. We shall apply Lemma B with $p = q + h$, where $p \le g$ and $q \ge 0$, thus in $g - h + 1$ cases altogether. The key observation is that $x^R \,\&\, \lfloor 2^p - 2^q \rfloor$ is independent of $x \,\&\, \lfloor 2^{p+s} - 2^{q+s} \rfloor$ whenever there are no indices j and k such that $0 \le j, k < h$ and $g - 1 - q - j = q + s + k$. The number of "bad" choices of q for which such

indices exist is at most $\frac{1}{2}(h+1)h^2 \le g-h$; therefore at least one "good" choice of q yields $|X_{pqr}| = 2^h$. But then Lemma B leads to a contradiction, because we obviously cannot have $2^h \le B_r \le 2^{(h-1)/2}$. ∎

Corollary M. *Multiplication by certain constants, modulo 2^n, requires $\Omega(\log n)$ steps in an n-bit broadword chain.*

Proof. In Hack 167 of the classic memorandum HAKMEM (M.I.T. A.I. Laboratory, 1972), Richard Schroeppel observed that the operations

$$t \leftarrow ((ax) \bmod 2^n) \,\&\, b, \quad y \leftarrow ((ct) \bmod 2^n) \gg (n-g) \tag{104}$$

compute $y = x^R$ whenever $n = g^2$ and $0 \le x < 2^g$, using the constants $a = (2^{n+g}-1)/(2^{g+1}-1)$, $b = 2^{g-1}(2^n-1)/(2^g-1)$, and $c = (2^{n-g}-1)/(2^{g-1}-1)$. (See exercise 123.) ∎

At this point the reader might well be thinking, "Okay, I agree that broadword chains sometimes have to be asymptotically long. But programmers needn't be shackled by such chains; we can use other techniques, like conditional branches or references to precomputed tables, which go beyond those restrictions."

Right. And we're in luck, because broadword theory can also be extended to more general models of computation. Consider, for example, the following idealization of an abstract reduced-instruction-set computer, called a *basic RAM*: The machine has n-bit registers $r_1, \ldots, r_l$, and n-bit memory words $\{M[0], \ldots, M[2^m-1]\}$. It can perform the instructions

$$\begin{gathered} r_i \leftarrow r_j \pm r_k, \quad r_i \leftarrow r_j \circ r_k, \quad r_i \leftarrow r_j \gg r_k, \quad r_i \leftarrow c, \\ r_i \leftarrow M[r_j \bmod 2^m], \quad M[r_j \bmod 2^m] \leftarrow r_i, \end{gathered} \tag{105}$$

where $\circ$ is any bitwise Boolean operator, and where r_k in the shift instruction is treated as a signed integer in two's complement notation. The machine is also able to branch if $r_i \le r_j$, treating r_i and r_j as unsigned integers. Its *state* is the entire contents of all registers and memory, together with a "program counter" that points to the current instruction. Its program begins in a designated state, which may include precomputed tables in memory, and with an n-bit input value x in register r_1. This initial state is called $Q(x,0)$, and $Q(x,t)$ denotes the state after t instructions have been performed. When the machine stops, r_1 will contain some n-bit value $f(x)$. Given a function $f(x)$, we want to find a lower bound on the least t such that r_1 is equal to $f(x)$ in state $Q(x,t)$, for $0 \le x < 2^n$.

Theorem R′. *Let $\epsilon = 2^{-e}$. A basic n-bit RAM with memory parameter $m \le n^{1-\epsilon}$ requires at least $\lg\lg n - e$ steps to evaluate the ruler function ρx, as $n \to \infty$.*

Proof. Let $n = 2^{2^{e+f}}$, so that $m \le 2^{2^{e+f}-2^f}$. Exercise 124 explains how an omniscient observer can construct a broadword chain from a certain class of inputs x, in such a way that each x causes the RAM to take the same branches, use the same shift amounts, and refer to the same memory locations. Our earlier methods can then be used to show that this chain has length $\ge f$. ∎

A skeptical reader may still object that Theorem R′ has no practical value, because $\lg\lg n$ never exceeds 5 in the real world. To this argument there is no rebuttal. But the following result is slightly more relevant:

HAKMEM
Schroeppel
abstract reduced-instruction-set computer
basic RAM
two's complement notation
program counter
ruler function

Theorem P′. *A basic n-bit RAM requires at least $\frac{1}{3}\lg g$ steps to compute the g-bit reversal x^R for $0 \le x < 2^g$, if $g \le n$ and*

$$\max(m, 1 + \lg n) \;<\; \frac{h+1}{2\lfloor \lg(h+1)\rfloor - 2}, \qquad h = \lfloor \sqrt[3]{g} \rfloor. \tag{106}$$

Proof. An argument like the proof of Theorem R′ appears in exercise 125. ▌

Lemma B and Theorems R, P, R′, P′ and their corollaries are due to A. Brodnik, P. B. Miltersen, and J. I. Munro, *Lecture Notes in Comp. Sci.* **1272** (1997), 426–439, based on earlier work of Miltersen in *Lecture Notes in Comp. Sci.* **1099** (1996), 442–451.

Many unsolved questions remain (see exercises 126–130). For example, does sideways addition require $\Omega(\log n)$ steps in an n-bit broadword chain? Can the parity function $(\nu x) \bmod 2$, or the majority function $(\nu x) > n/2$, be computed broadwordwise in $O(\log\log n)$ steps — or even perhaps in constant time?

An application to directed graphs. Now let's use some of what we've learned, by implementing a simple algorithm. Given a digraph on a set of vertices V, we write $u \to v$ when there's an arc from u to v. The *reachability problem* is to find all vertices that lie on oriented paths beginning in a specified set $Q \subseteq V$; in other words, we seek the set

$$R \;=\; \{v \mid u \to^* v \text{ for some } u \in Q\}, \tag{107}$$

where $u \to^* v$ means that there is a sequence of t arcs

$$u = u_0 \to u_1 \to \cdots \to u_t = v, \qquad \text{for some } t \ge 0. \tag{108}$$

This problem arises frequently in practice. For example, we encountered it in Section 2.3.5 when marking all elements of Lists that are not "garbage."

If the number of vertices is small, say $|V| \le 64$, we may want to approach the reachability problem in quite a different way than we did before, by working directly with subsets of vertices. Let

$$S[u] \;=\; \{v \mid u \to v\} \tag{109}$$

be the set of successors of vertex u, for all $u \in V$. Then the following algorithm is almost completely different from Algorithm 2.3.5E, yet it solves the same abstract problem:

Algorithm R (*Reachability*). Given a directed graph, represented by the successor sets $S[u]$ in (109), this algorithm computes the elements R that are reachable from a given set Q.

R1. [Initialize.] Set $R \leftarrow Q$ and $X \leftarrow \emptyset$. (In the following steps, X is the subset of vertices $u \in R$ for which we've looked at $S[u]$.)

R2. [Done?] If $X = R$, the algorithm terminates.

R3. [Examine another vertex.] Let u be an element of $R \setminus X$. Set $X \leftarrow X \cup \{u\}$, $R \leftarrow R \cup S[u]$, and return to step R2. ▌

The algorithm is correct because (i) every element placed into R is reachable; (ii) every reachable element u_j in (108) is present in R, by induction on j; and (iii) termination eventually occurs, because step R3 always increases $|X|$.

Brodnik
Miltersen
Munro
sideways addition
parity function
majority function
graph algorithms+
reachability problem
oriented paths
transitive closure
notation $u \to^* v$
garbage

adjacency matrix
little-endian order
ruler function
encoding of ternary data
representing three states with two bits+++
mapping three items into two-bit codes+++

To implement Algorithm R we will assume that $V = \{0, 1, \ldots, n-1\}$, with $n \le 64$. The set X is conveniently represented by the integer $\sigma(X) = \sum\{2^u \mid u \in X\}$, and the same convention works nicely for the other sets Q, R, and $S[u]$. Notice that the bits of $S[0]$, $S[1]$, $\ldots$, $S[n-1]$ are essentially the *adjacency matrix* of the given digraph, as explained in Section 7, but in little-endian order: The "diagonal" elements, which tell us whether or not $u \in S[u]$, go from right to left. For example, if $n = 3$ and the arcs are $\{0\to 0, 0\to 1, 1\to 0, 2\to 0\}$, we have $S[0] = (011)_2$ and $S[1] = S[2] = (001)_2$, while the adjacency matrix is $\left(\begin{smallmatrix}110\\100\\100\end{smallmatrix}\right)$.

Step R3 allows us to choose any element of $R\setminus X$, so we use the ruler function $u \leftarrow \rho(\sigma(R) - \sigma(X))$ to choose the smallest. The bitwise operations require no further trickery when we adapt the algorithm to `MMIX`:

Program R (*Reachability*). The input set Q is given in register `q`, and each successor set $S[u]$ appears in octabyte $\mathrm{M}_8[\texttt{suc} + 8u]$. The output set R will appear in register `r`; other registers `s`, `t`, `tt`, `u`, and `x` hold intermediate results.

1H	`SET`	`r,q`	1	<u>*R1. Initialize.*</u> $\texttt{r} \leftarrow \sigma(Q)$.
	`SET`	`x,0`	1	$\texttt{x} \leftarrow \sigma(\emptyset)$.
	`JMP`	`2F`	1	To R2.
3H	`SUBU`	`tt,t,1`	$\lvert R\rvert$	<u>*R3. Examine another vertex.*</u> $\texttt{tt} \leftarrow \texttt{t} - 1$.
	`SADD`	`u,tt,t`	$\lvert R\rvert$	$u \leftarrow \rho(\texttt{t})$ [see (46)].
	`SLU`	`s,u,3`	$\lvert R\rvert$	$\texttt{s} \leftarrow 8u$.
	`LDOU`	`s,suc,s`	$\lvert R\rvert$	$\texttt{s} \leftarrow \sigma(S[u])$.
	`ANDN`	`tt,t,tt`	$\lvert R\rvert$	$\texttt{tt} \leftarrow \texttt{t} \mathbin{\&} {\sim}\texttt{tt} = 2^u$.
	`OR`	`x,x,tt`	$\lvert R\rvert$	$X \leftarrow X \cup \{u\}$; that is, $\texttt{x} \leftarrow \texttt{x} \mid 2^u$, since $\texttt{x} = \sigma(X)$.
	`OR`	`r,r,s`	$\lvert R\rvert$	$R \leftarrow R \cup S[u]$; that is, $\texttt{r} \leftarrow \texttt{r} \mid \texttt{s}$, since $\texttt{r} = \sigma(R)$.
2H	`SUBU`	`t,r,x`	$\lvert R\rvert + 1$	<u>*R2. Done?*</u> $\texttt{t} \leftarrow \texttt{r} - \texttt{x} = \sigma(R \setminus X)$, since $X \subseteq R$.
	`PBNZ`	`t,3B`	$\lvert R\rvert + 1$	To R3 if $R \ne X$. ▮

The total running time is $(\mu + 9\upsilon)|R| + 7\upsilon$. By contrast, exercise 131 implements Algorithm R with linked lists; the overall execution time then grows to $(3S + 4|R| - 2|Q| + 1)\mu + (5S + 12|R| - 5|Q| + 4)\upsilon$, where $S = \sum_{u\in R} |S[u]|$. (But of course that program is also able to handle graphs with millions of vertices.)

Exercise 132 presents another instructive algorithm where bitwise operations work nicely on not-too-large graphs.

Application to data representation. Computers are binary, but (alas?) the world isn't. We often must find a way to encode nonbinary data into 0s and 1s. One of the most common problems of this sort is to choose an efficient representation for items that can be in exactly three different states.

Suppose we know that $x \in \{a, b, c\}$, and we want to represent x by two bits $x_l x_r$. We could, for example, map $a \mapsto 00$, $b \mapsto 01$, and $c \mapsto 10$. But there are many other possibilities — in fact, 4 choices for a, then 3 choices for b, and 2 for c, making 24 altogether. Some of these mappings might be much easier to deal with than others, depending on what we want to do with x.

Given two elements $x, y \in \{a, b, c\}$, we typically want to compute $z = x \circ y$, for some binary operation $\circ$. If $x = x_l x_r$ and $y = y_l y_r$ then $z = z_l z_r$, where

$$z_l = f_l(x_l, x_r, y_l, y_r) \qquad \text{and} \qquad z_r = f_r(x_l, x_r, y_l, y_r); \tag{110}$$

these Boolean functions f_l and f_r of four variables depend on $\circ$ and the chosen representation. We seek a representation that makes f_l and f_r easy to compute.

Suppose, for example, that $\{a, b, c\} = \{-1, 0, +1\}$ and that $\circ$ is multiplication. If we decide to use the natural mapping $x \mapsto x \bmod 3$, namely

$$0 \mapsto 00, \qquad +1 \mapsto 01, \qquad -1 \mapsto 10, \tag{111}$$

so that $x = x_r - x_l$, then the truth tables for f_l and f_r are respectively

$$f_l \leftrightarrow 000{*}001{*}010{*}{*}{*}{*}{*} \qquad \text{and} \qquad f_r \leftrightarrow 000{*}010{*}001{*}{*}{*}{*}{*}. \tag{112}$$

(There are seven "don't-cares," for cases where $x_l x_r = 11$ and/or $y_l y_r = 11$.) The methods of Section 7.1.2 tell us how to compute z_l and z_r optimally, namely

$$z_l = (x_l \oplus y_l) \wedge (x_r \oplus y_r), \qquad z_r = (x_l \oplus y_r) \wedge (x_r \oplus y_l); \tag{113}$$

unfortunately the functions f_l and f_r in (112) are independent, in the sense that they cannot both be evaluated in fewer than $C(f_l) + C(f_r) = 6$ steps.

On the other hand the somewhat less natural mapping scheme

$$+1 \mapsto 00, \qquad 0 \mapsto 01, \qquad -1 \mapsto 10 \tag{114}$$

leads to the transformation functions

$$f_l \leftrightarrow 001{*}000{*}100{*}{*}{*}{*}{*} \qquad \text{and} \qquad f_r \leftrightarrow 010{*}111{*}010{*}{*}{*}{*}{*}, \tag{115}$$

and three operations now suffice to do the desired evaluation:

$$z_l = x_r \vee y_r, \qquad z_r = (x_l \oplus y_l) \wedge \bar{z}_l. \tag{116}$$

Is there an easy way to discover such improvements? Fortunately we don't need to try all 24 possibilities, because many of them are basically alike. For example, the mapping $x \mapsto x_r x_l$ is equivalent to $x \mapsto x_l x_r$, because the new representation $x'_l x'_r = x_r x_l$ obtained by swapping coordinates makes

$$f'_l(x'_l, x'_r, y'_l, y'_r) = z'_l = z_r = f_r(x_l, x_r, y_l, y_r);$$

the new transformation functions f'_l and f'_r defined by

$$f'_l(x_l, x_r, y_l, y_r) = f_r(x_r, x_l, y_r, y_l), \quad f'_r(x_l, x_r, y_l, y_r) = f_l(x_r, x_l, y_r, y_l) \tag{117}$$

have the same complexity as f_l and f_r. Similarly we can complement a coordinate, letting $x'_l x'_r = \bar{x}_l x_r$; then the transformation functions turn out to be

$$f'_l(x_l, x_r, y_l, y_r) = \bar{f}_l(\bar{x}_l, x_r, \bar{y}_l, y_r), \quad f'_r(x_l, x_r, y_l, y_r) = \bar{f}_r(\bar{x}_l, x_r, \bar{y}_l, y_r), \tag{118}$$

and again the complexity is essentially unchanged.

Repeated use of swapping and/or complementation leads to eight mappings that are equivalent to any given one. So the 24 possibilities reduce to only three, which we shall call classes I, II, and III:

$$\begin{array}{l} \\ a \mapsto \\ b \mapsto \\ c \mapsto \end{array} \begin{array}{c} \text{Class I} \\ \overbrace{\hphantom{00\ 01\ 10\ 11\ 00\ 10\ 01\ 11}} \\ 00\ 01\ 10\ 11\ 00\ 10\ 01\ 11 \\ 01\ 00\ 11\ 10\ 10\ 00\ 11\ 01 \\ 10\ 11\ 00\ 01\ 01\ 11\ 00\ 10 \end{array} \begin{array}{c} \text{Class II} \\ \overbrace{\hphantom{00\ 01\ 10\ 11\ 00\ 10\ 01\ 11}} \\ 00\ 01\ 10\ 11\ 00\ 10\ 01\ 11 \\ 01\ 00\ 11\ 10\ 10\ 00\ 11\ 01 \\ 11\ 10\ 01\ 00\ 11\ 01\ 10\ 00 \end{array} \begin{array}{c} \text{Class III} \\ \overbrace{\hphantom{00\ 01\ 10\ 11\ 00\ 10\ 01\ 11}} \\ 00\ 01\ 10\ 11\ 00\ 10\ 01\ 11; \\ 11\ 10\ 01\ 00\ 11\ 01\ 10\ 00; \\ 01\ 00\ 11\ 10\ 10\ 00\ 11\ 01. \end{array} \tag{119}$$

multiplication of signed bits+
signed bits, representation of
don't-cares
2-cube equivalence

one-to-many mapping
don't-cares
2-cube equivalence
don't-cares

To choose a representation we need consider only one representative of each class. For example, if $a = +1$, $b = 0$, and $c = -1$, representation (111) belongs to class II, and (114) belongs to class I. Class III turns out to have cost 3, like class I. So it appears that representation (114) is as good as any, with z computed by (116), for the 3-element multiplication problem we've been studying.

Appearances can, however, be deceiving, because we need not map $\{a, b, c\}$ into *unique* two-bit codes. Consider the one-to-many mapping

$$+1 \mapsto 00, \qquad 0 \mapsto 01 \text{ or } 11, \qquad -1 \mapsto 10, \tag{120}$$

where both 01 and 11 are allowed as representations of zero. The truth tables for f_l and f_r are now quite different from (112) and (115), because all inputs are legal but some outputs can be arbitrary:

$$f_l \leftrightarrow 0{*}1{*}{*}{*}{*}{*}1{*}0{*}{*}{*}{*}{*} \qquad \text{and} \qquad f_r \leftrightarrow 0101111101011111. \tag{121}$$

And in fact, this approach needs just two operations, instead of the three in (116):

$$z_l = x_l \oplus y_l, \qquad z_r = x_r \vee y_r. \tag{122}$$

A moment's thought shows that indeed, these operations obviously yield the product $z = x \cdot y$ when the three elements $\{+1, 0, -1\}$ are represented as in (120).

Such nonunique mappings add 36 more possibilities to the 24 that we had before. But again, they reduce under "2-cube equivalence" to a small number of equivalence classes. First there are three classes that we call IV_a, IV_b, and IV_c, depending on which element has an ambiguous representation:

$$\begin{array}{llll} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class IV}_a} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class IV}_b} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class IV}_c} \\ a \mapsto & 0{*}\ 0{*}\ 1{*}\ 1{*}\ {*}0\ {*}0\ {*}1\ {*}1 & 11\ 10\ 01\ 00\ 11\ 01\ 10\ 00 & 10\ 11\ 00\ 01\ 01\ 11\ 00\ 10; \\ b \mapsto & 10\ 11\ 00\ 01\ 01\ 11\ 00\ 10 & 0{*}\ 0{*}\ 1{*}\ 1{*}\ {*}0\ {*}0\ {*}1\ {*}1 & 11\ 10\ 01\ 00\ 11\ 01\ 10\ 00; \\ c \mapsto & 11\ 10\ 01\ 00\ 11\ 01\ 10\ 00 & 10\ 11\ 00\ 01\ 01\ 11\ 00\ 10 & 0{*}\ 0{*}\ 1{*}\ 1{*}\ {*}0\ {*}0\ {*}1\ {*}1. \end{array} \tag{123}$$

(Representation (120) belongs to Class IV_b. Classes IV_a and IV_c don't work well for $z = x \cdot y$.) Then there are three further classes with only four mappings each:

$$\begin{array}{llll} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class V}_a} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class V}_b} & \overbrace{\qquad\qquad\qquad\qquad}^{\text{Class V}_c} \\ a \mapsto & tt\quad t\bar{t}\quad t\bar{t}\quad tt & 10\quad 11\quad 00\quad 01 & 01\quad 00\quad 11\quad 10; \\ b \mapsto & 01\quad 00\quad 11\quad 10 & tt\quad t\bar{t}\quad t\bar{t}\quad tt & 10\quad 11\quad 00\quad 01; \\ c \mapsto & 10\quad 11\quad 00\quad 01 & 01\quad 00\quad 11\quad 10 & tt\quad t\bar{t}\quad t\bar{t}\quad tt. \end{array} \tag{124}$$

These classes are a bit of a nuisance, because the indeterminacy in their truth tables cannot be expressed simply in terms of don't-cares as we did in (121). For example, if we try

$$+1 \mapsto 00 \text{ or } 11, \qquad 0 \mapsto 01, \qquad -1 \mapsto 10, \tag{125}$$

which is the first mapping in Class V_a, there are binary variables $pqrst$ such that

$$f_l \leftrightarrow p01q000010r1s01t \qquad \text{and} \qquad f_r \leftrightarrow p10q111101r0s10t. \tag{126}$$

Furthermore, mappings of classes V_a, V_b, and V_c almost never turn out to be better than the mappings of the other six classes (see exercise 138). Still, representatives of all nine classes must be examined before we can be sure that an optimal mapping has been found.

In practice we often want to perform several different operations on ternary-valued variables, not just a single operation like multiplication. For example, we might want to compute $\max(x, y)$ as well as $x \cdot y$. With representation (120), the best we can do is $z_l = x_l \wedge y_l$, $z_r = (x_l \wedge y_r) \vee (x_r \wedge (y_l \vee y_r))$; but the "natural" mapping (111) now shines, with $z_l = x_l \wedge y_l$, $z_r = x_r \vee y_r$. Class III turns out to have cost 4; other classes are inferior. To choose between classes II, III, and IV_b in this case, we need to know the relative frequencies of $x \cdot y$ and $\max(x, y)$. And if we add $\min(x, y)$ to the mix, classes II, III, and IV_b compute it with the respective costs 2, 5, 5; hence (111) looks better yet.

The ternary max and min operations arise also in other contexts, such as the three-valued logic developed by Jan Łukasiewicz in 1917. [See his *Selected Works*, edited by L. Borkowski (1970), 84–88, 153–178.] Consider the logical values "true," "false," and "maybe," denoted respectively by 1, 0, and $*$. Łukasiewicz defined the three basic operations of conjunction, disjunction, and implication on these values by specifying the tables

$$
\underbrace{\begin{array}{c|ccc|} & 0 & * & 1 \\ \hline 0 & 0 & 0 & 0 \\ * & 0 & * & * \\ 1 & 0 & * & 1 \\ \hline \end{array}}_{x \wedge y}, \qquad
\underbrace{\begin{array}{c|ccc|} & 0 & * & 1 \\ \hline 0 & 0 & * & 1 \\ * & * & * & 1 \\ 1 & 1 & 1 & 1 \\ \hline \end{array}}_{x \vee y}, \qquad
\underbrace{\begin{array}{c|ccc|} & 0 & * & 1 \\ \hline 0 & 1 & 1 & 1 \\ * & * & 1 & 1 \\ 1 & 0 & * & 1 \\ \hline \end{array}}_{x \Rightarrow y}. \qquad (127)
$$

(rows indexed by x, columns by y)

For these operations the methods above show that the binary representation

$$0 \mapsto 00, \qquad * \mapsto 01, \qquad 1 \mapsto 11 \qquad (128)$$

works well, because we can compute the logical operations thus:

$$
\begin{aligned}
x_l x_r \wedge y_l y_r &= (x_l \wedge y_l)(x_r \wedge y_r), \qquad x_l x_r \vee y_l y_r = (x_l \vee y_l)(x_r \vee y_r), \\
x_l x_r \Rightarrow y_l y_r &= ((x_r \oplus y_r) \wedge \neg(\bar{x}_l \wedge y_r))(x_l \wedge \bar{y}_r).
\end{aligned} \qquad (129)
$$

Of course x need not be an isolated ternary value in this discussion; we often want to deal with ternary *vectors* $x = x_1 x_2 \ldots x_n$, where each x_j is either a, b, or c. Such ternary vectors are conveniently represented by two binary vectors

$$x_l = x_{1l} x_{2l} \ldots x_{nl} \qquad \text{and} \qquad x_r = x_{1r} x_{2r} \ldots x_{nr}, \qquad (130)$$

where $x_j \mapsto x_{jl} x_{jr}$ as above. We could also pack the ternary values into two-bit fields of a single vector,

$$x = x_{1l} x_{1r} x_{2l} x_{2r} \ldots x_{nl} x_{nr}; \qquad (131)$$

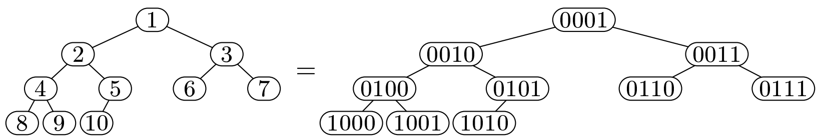
that would work fine if, say, we're doing Łukasiewicz logic with the operations $\wedge$ and $\vee$ but not $\Rightarrow$. Usually, however, the two-vector approach of (130) is better, because it lets us do bitwise calculations without shifting and masking.

max
min
three-valued logic
Łukasiewicz
Borkowski
modal logic
maybe
conjunction
disjunction
implication
groupoids, multiplication tables
ternary vectors
pack
masking: ANDing with a mask

Applications to data structures. Bitwise operations offer many efficient ways to represent elements of data and the relationships between them. For example, chess-playing programs often use a "bit board" to represent the positions of pieces (see exercise 143).

In Chapter 8 we shall discuss an important data structure developed by Peter van Emde Boas for representing a dynamically changing subset of integers between 0 and N. Insertions, deletions, and other operations such as "find the largest element less than x" can be done in $O(\log\log N)$ steps with his methods; the general idea is to organize the full structure recursively as $\sqrt{N}$ substructures for subsets of intervals of size $\sqrt{N}$, together with an auxiliary structure that tells which of those intervals are occupied. [See *Information Processing Letters* **6** (1977), 80–82; also P. van Emde Boas, R. Kaas, and E. Zijlstra, *Math. Systems Theory* **10** (1977), 99–127.] Bitwise operations make those computations fast.

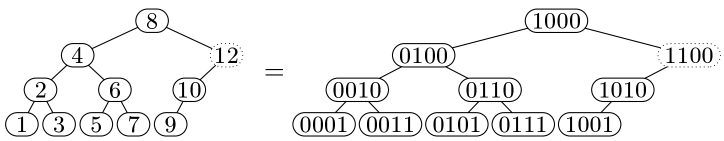
Hierarchical data can sometimes be arranged so that the links between elements are implicit rather than explicit. For example, we studied "heaps" in Section 5.2.3, where n elements of a sequential array implicitly have a binary tree structure like



(132)

when, say, $n = 10$. (Node numbers are shown here both in decimal and binary notation.) There is no need to store pointers in memory to relate node j of a heap to its parent (which is node $j \gg 1$ if $j \neq 1$), or to its sibling (which is node $j \oplus 1$ if $j \neq 1$), or to its children (which are nodes $j \ll 1$ and $(j \ll 1) + 1$ if those numbers don't exceed n), because a simple calculation leads directly from j to any desired neighbor.

Similarly, a *sideways heap* provides implicit links for another useful family of n-node binary tree structures, typified by



(133)

when $n = 10$. (We sometimes need to go beyond n when moving from a node to its parent, as in the path from 10 to 12 to 8 shown here.) Heaps and sideways heaps can both be regarded as nodes 1 to n of *infinite* binary tree structures: The heap with $n = \infty$ is rooted at node 1 and has no leaves; by contrast, the sideways heap with $n = \infty$ has infinitely many leaves 1, 3, 5, ..., but no root(!).

The leaves of a sideways heap are the odd numbers, and their parents are the odd multiples of 2. The grandparents of leaves, similarly, are the odd multiples of 4; and so on. Thus the ruler function ρj tells how high node j is above leaf level.

The parent of node j in the infinite sideways heap is easily seen to be node

$$(j - k) \mid (k \ll 1), \qquad \text{where } k = j \,\&\, {-j}; \tag{134}$$

chess
bit board
Emde Boas
van Emde Boas
Kaas
Zijlstra
implicit data structures–
heaps
sibling
sideways heap
binary tree structures
ruler function

this quantity is j rounded to the nearest multiple of $2^{1+\rho j}$. And the children are

$$j - (k \gg 1) \qquad \text{and} \qquad j + (k \gg 1) \tag{135}$$

when j is even. In general the descendants of node j form a closed interval

$$[j - 2^{\rho j} + 1 \mathinner{..} j + 2^{\rho j} - 1], \tag{136}$$

arranged as a complete binary tree of $2^{1+\rho j}$ nodes. The ancestor of node j at height h is node

$$(j \mid (1 \ll h)) \mathbin{\&} -(1 \ll h) = ((j \gg h) \mid 1) \ll h \tag{137}$$

when $h \geq \rho j$. Notice that the symmetric order of the nodes, also called inorder, is just the natural order 1, 2, 3, ...

Dov Harel noted these properties in his Ph.D. thesis (U. of California, Irvine, 1980), and observed that the *nearest* common ancestor of any two nodes of a sideways heap can also be easily calculated. Indeed, if node l is the nearest common ancestor of nodes i and j, where $i \leq j$, there is a remarkable identity

$$\rho l \;=\; \max\{\rho x \mid i \leq x \leq j\} \;=\; \lambda(j \mathbin{\&} -i), \tag{138}$$

which relates the ρ and λ functions. (See exercise 146.) We can therefore use formula (137) with $h = \lambda(j \mathbin{\&} -i)$ to calculate l.

Subtle extensions of this approach lead to an asymptotically efficient algorithm that finds nearest common ancestors in *any* oriented forest whose arcs grow dynamically [D. Harel and R. E. Tarjan, *SICOMP* **13** (1984), 338–355]. Baruch Schieber and Uzi Vishkin [*SICOMP* **17** (1988), 1253–1262] subsequently discovered a much simpler way to compute nearest common ancestors in an arbitrary (but fixed) oriented forest, using an attractive and instructive blend of bitwise and algorithmic techniques that we shall consider next.

Recall that an oriented forest with m trees and n vertices is an acyclic digraph with $n - m$ arcs. There is at most one arc from each vertex; the vertices with out-degree zero are the roots of the trees. We say that v is the *parent* of u when $u \to v$, and v is an *ancestor* of u when $u \to^* v$. Two vertices have a common ancestor if and only if they belong to the same tree. Vertex w is called the nearest common ancestor of u and v when we have

$$u \to^* z \text{ and } v \to^* z \quad \text{if and only if} \quad w \to^* z. \tag{139}$$

Schieber and Vishkin preprocess the given forest, mapping its vertices into a sideways heap S of size n by computing three quantities for each vertex v:

πv, the rank of v in preorder $(1 \leq \pi v \leq n)$;

βv, a node of the sideways heap S $(1 \leq \beta v \leq n)$;

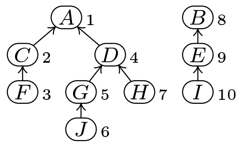
αv, a $(1 + \lambda n)$-bit routing code $(1 \leq \alpha v < 2^{1+\lambda n})$.

If $u \to v$ we have $\pi u > \pi v$ by the definition of preorder. Node βv is defined to be the nearest common ancestor of all sideways-heap nodes πu such that v is an ancestor of vertex u. And we define

$$\alpha v \;=\; \sum \{2^{\rho \beta w} \mid v \to^* w\}. \tag{140}$$

rounded
complete binary tree
symmetric order
inorder
Harel
lowest common ancestor, see Nearest comm
λ
Harel
Tarjan
Schieber
Vishkin
oriented forest
acyclic digraph
ancestor
reachability
transitive closure
nearest common ancestor
preorder++

For example, here's an oriented forest with ten vertices and two trees:



(141)

Each node has been labeled with its preorder rank, from which we can compute the β and α codes:

$v =$	A	B	C	D	E	F	G	H	I	J
$\pi v =$	0001	1000	0010	0100	1001	0011	0101	0111	1010	0110
$\beta v =$	0100	1000	0010	0100	1010	0011	0110	0111	1010	0110
$\alpha v =$	0100	1000	0110	0100	1010	0111	0110	0101	1010	0110

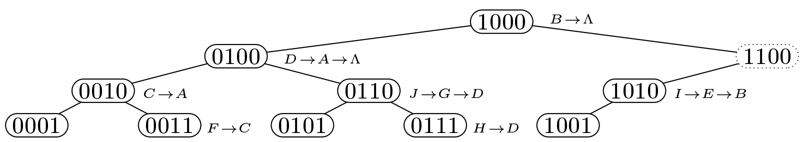
Notice that, for instance, $\beta A = 4 = 0100$ because the preorder ranks of the descendants of A are $\{1,2,3,4,5,6,7\}$. And $\alpha H = 0101$ because the ancestors of H have β codes $\{\beta H, \beta D, \beta A\} = \{0111, 0100\}$. One can prove without difficulty that the mapping $v \mapsto \beta v$ satisfies the following key properties:

i) If $u \to v$ in the forest, then βu is a descendant of βv in S.

ii) If several vertices have the same value of βv, they form a path in the forest.

Property (ii) holds because exactly one child u of v has $\beta u = \beta v$ when $\beta v \neq \pi v$.

Now let's imagine placing every vertex v of the forest into node βv of S:



(142)

If k vertices map into node j, we can arrange them into a path

$$v_0 \to v_1 \to \cdots \to v_{k-1} \to v_k, \qquad \text{where } \beta v_0 = \beta v_1 = \cdots = \beta v_{k-1} = j. \quad (143)$$

These paths are illustrated in (142); for example, $J \to G \to D$ is a path in (141), and '$J \to G \to D$' appears with node $0110 = \beta J = \beta G$.

The preprocessing algorithm also computes a table τj for all nodes j of S, containing pointers to the vertices v_k at the tail ends of (143):

$j =$	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010
$\tau j =$	Λ	A	C	Λ	Λ	D	D	Λ	Λ	B

Exercise 149 shows that all four tables πv, βv, αv, and τj can be prepared in $O(n)$ steps. And once those tables are ready, they contain just enough information to identify the nearest common ancestor of any two given vertices quickly:

Algorithm V (*Nearest common ancestors*). Suppose πv, βv, αv, and τj are known for all n vertices v of an oriented forest, and for $1 \le j \le n$. A dummy vertex Λ is also assumed to be present, with $\pi\Lambda = \beta\Lambda = \alpha\Lambda = 0$. This algorithm computes the nearest common ancestor z of any given vertices x and y, returning $z = \Lambda$ if x and y belong to different trees. We assume that the values $\lambda j = \lfloor \lg j \rfloor$ have been precomputed for $1 \le j \le n$, and that $\lambda 0 = \lambda n$.

V1. [Find common height.] If $\beta x \le \beta y$, set $h \leftarrow \lambda(\beta y \,\&\, -\beta x)$; otherwise set $h \leftarrow \lambda(\beta x \,\&\, -\beta y)$. (See (138).)

V2. [Find true height.] Set $k \leftarrow \alpha x \,\&\, \alpha y \,\&\, -(1 \ll h)$, then $h \leftarrow \lambda(k \,\&\, -k)$.

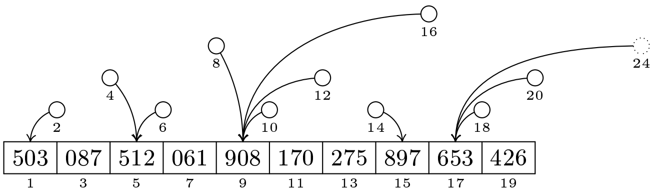
V3. [Find βz.] Set $j \leftarrow ((\beta x \gg h) \mid 1) \ll h$. (Now $j = \beta z$, if $z \ne \Lambda$.)

V4. [Find $\hat{x}$ and $\hat{y}$.] (We now seek the lowest ancestors of x and y in node j.) If $j = \beta x$, set $\hat{x} = x$; otherwise set $l \leftarrow \lambda(\alpha x \,\&\, ((1 \ll h) - 1))$ and $\hat{x} = \tau(((\beta x \gg l) \mid 1) \ll l)$. Similarly, if $j = \beta y$, set $\hat{y} = y$; otherwise set $l \leftarrow \lambda(\alpha y \,\&\, ((1 \ll h) - 1))$ and $\hat{y} = \tau(((\beta y \gg l) \mid 1) \ll l)$.

V5. [Find z.] Set $z \leftarrow \hat{x}$ if $\pi\hat{x} \le \pi\hat{y}$, otherwise $z \leftarrow \hat{y}$. ▮

These artful dodges obviously exploit (137); exercise 152 explains why they work.

Sideways heaps can also be used to implement an interesting type of priority queue that J. Katajainen and F. Vitale call a "navigation pile," illustrated here for $n = 10$:



(144)

Data elements go into the leaf positions 1, 3, ..., $2n - 1$ of the sideways heap; they can be many bits wide, and they can appear in any order. By contrast, each branch position 2, 4, 6, ... contains a pointer to its largest descendant. And the novel point is that these pointers take up almost no extra space — fewer than two bits per item of data, on average — because only one bit is needed for pointers 2, 6, 10, ..., only two bits for pointers 4, 12, 20, ..., and only ρj for pointer j in general. (See exercise 153.) Thus the navigation pile requires very little memory, and it behaves nicely with respect to cache performance on a typical computer.

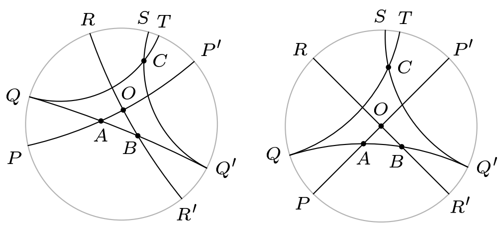


Fig. 13. Two views of five lines in the hyperbolic plane.

***Cells in the hyperbolic plane.** Hyperbolic geometry suggests an instructive implicit data structure that has a rather different flavor. The *hyperbolic plane* is a fascinating example of non-Euclidean geometry that is conveniently viewed by projecting its points into the interior of a circle. Its straight lines then become circular arcs, which meet the rim at right angles. For example, the lines PP', QQ', and RR' in Fig. 13 intersect at points O, A, B, and those points form a triangle. Lines SQ' and QQ' are *parallel*: They never touch, but their points get closer and closer together. Line QT is also parallel to QQ'.

priority queue
Katajainen
Vitale
navigation pile
cache
hyperbolic plane–
non-Euclidean geometry

We get different views by focusing on different center points. For example, the second view in Fig. 13 puts O smack in the center. Notice that if a line passes through the very center, it remains straight after being projected; such diameter-spanning chords are the special case of a "circular arc" whose radius is infinite.

Most of Euclid's axioms for plane geometry remain valid in the hyperbolic plane. For example, exactly one line passes through any two distinct points; and if point A lies on line PP' there's exactly one line QQ' such that angle PAQ has any given value θ, for $0 < \theta < 180°$. But Euclid's famous fifth postulate does *not* hold: If point C is *not* on line QQ', there always are exactly *two* lines through C that are parallel to QQ'. Furthermore there are many pairs of lines, like RR' and SQ' in Fig. 13, that are totally disjoint or *ultraparallel*, in the sense that their points never become arbitrarily close. [These properties of the hyperbolic plane were discovered by G. Saccheri in the early 1700s, and made rigorous by N. I. Lobachevsky, J. Bolyai, and C. F. Gauss a century later.]

Quantitatively speaking, when points are projected onto the unit disk $|z|<1$, the arc that meets the circle at $e^{i\theta}$ and $e^{-i\theta}$ has center at $\sec\theta$ and radius $\tan\theta$. The actual distance between two points whose projections are z and z' is $d(z,z') = \ln(|1-\bar{z}z'| + |z-z'|) - \ln(|1-\bar{z}z'| - |z-z'|)$. Thus objects far from the center appear dramatically shrunken when we see them near the circle's rim.

The sum of the angles of a hyperbolic triangle is always *less* than 180°. For example, the angles at O, A, and B in Fig. 13 are respectively 90°, 45°, and 36°. Ten such 36°-45°-90° triangles can be placed together to make a regular pentagon with 90° angles at each corner. And four such pentagons fit snugly together at their corners, allowing us to tile the entire hyperbolic plane with right regular pentagons (see Fig. 14). The edges of these pentagons form an interesting family of lines, every two of which are either ultraparallel or perpendicular; so we have a grid structure analogous to the unit squares of the ordinary plane. We call it the *pentagrid*, because each cell now has five neighbors instead of four.

There's a nice way to navigate in the pentagrid using Fibonacci numbers, based on ideas of Maurice Margenstern [see F. Herrmann and M. Margenstern, *Theoretical Comp. Sci.* **296** (2003), 345–351]. Instead of the ordinary Fibonacci sequence $\langle F_n\rangle$, however, we shall use the *negaFibonacci* numbers $\langle F_{-n}\rangle$, namely

$$F_{-1}=1,\ F_{-2}=-1,\ F_{-3}=2,\ F_{-4}=-3,\ \ldots,\ F_{-n}=(-1)^{n-1}F_n. \qquad (145)$$

Exercise 1.2.8–34 introduced the Fibonacci number system, in which every nonnegative integer x can be written uniquely in the form

$$x = F_{k_1} + F_{k_2} + \cdots + F_{k_r}, \qquad \text{where } k_1 \succ\!\succ k_2 \succ\!\succ \cdots \succ\!\succ k_r \succ\!\succ 0; \qquad (146)$$

here '$j \succ\!\succ k$' means '$j \ge k+2$'. But there's also a *negaFibonacci number system*, which suits our purposes better: *Every integer x, whether positive, negative, or zero, can be written uniquely in the form*

$$x = F_{k_1} + F_{k_2} + \cdots + F_{k_r}, \qquad \text{where } k_1 \prec\!\prec k_2 \prec\!\prec \cdots \prec\!\prec k_r \prec\!\prec 1. \qquad (147)$$

For example, $4 = 5 - 1 = F_{-5} + F_{-2}$ and $-2 = -3 + 1 = F_{-4} + F_{-1}$. This representation can conveniently be expressed as a binary code $\alpha = \ldots a_3a_2a_1$,

Euclid
ultraparallel
Saccheri
Lobachevsky
Bolyai
Gauss
tile
ultraparallel
perpendicular
grid structure
tessellation
pentagrid
Fibonacci numbers
Margenstern
Herrmann
negaFibonacci
Fibonacci number system
negaFibonacci number system

negadecimal system
2-adic
magic mask

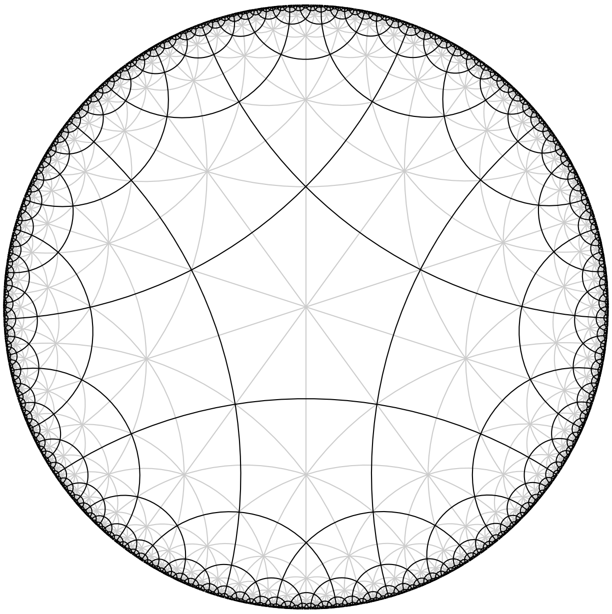


Fig. 14. The pentagrid, in which identical pentagons tile the hyperbolic plane.

A circular regular tiling, confined on all sides by infinitely small shapes, is really wonderful.
— M. C. ESCHER, letter to George Escher (9 November 1958)

standing for $N(\alpha) = \sum_k a_k F_{-k}$, with no two 1s in a row. For example, here are the negaFibonacci representation codes of all integers between -14 and $+15$:

$-14 = 10010100$	$-8 = 100000$	$-2 = 1001$	$4 = 10010$	$10 = 1001000$
$-13 = 10010101$	$-7 = 100001$	$-1 = 10$	$5 = 10000$	$11 = 1001001$
$-12 = 101010$	$-6 = 100100$	$0 = 0$	$6 = 10001$	$12 = 1000010$
$-11 = 101000$	$-5 = 100101$	$1 = 1$	$7 = 10100$	$13 = 1000000$
$-10 = 101001$	$-4 = 1010$	$2 = 100$	$8 = 10101$	$14 = 1000001$
$-9 = 100010$	$-3 = 1000$	$3 = 101$	$9 = 1001010$	$15 = 1000100$

As in the negadecimal system (see 4.1–(6) and (7)), we can tell whether x is negative or not by seeing if its representation has an even or odd number of digits.

The predecessor $\alpha-$ and successor $\alpha+$ of any negaFibonacci binary code α can be computed recursively by using the rules

$$(\alpha 01)- = \alpha 00,\quad (\alpha 000)- = \alpha 010,\quad (\alpha 100)- = \alpha 001,\quad (\alpha 10)- = (\alpha-)01,$$
$$(\alpha 10)+ = \alpha 00,\quad (\alpha 00)+ = \alpha 01,\quad (\alpha 1)+ = (\alpha-)0. \qquad (148)$$

(See exercise 154.) But ten elegant 2-adic steps do the calculation directly:

$$\begin{aligned} &y \leftarrow x \oplus \bar{\mu}_0,\ z \leftarrow y \oplus (y \pm 1),\ \text{where } x = (\alpha)_2;\\ &z \leftarrow z \mid (x \,\&\, (z \ll 1));\\ &w \leftarrow x \oplus z \oplus ((z+1) \gg 2);\ \text{then } w = (\alpha\pm)_2. \end{aligned} \qquad (149)$$

We just use $y-1$ in the top line to get the predecessor, $y+1$ to get the successor.

And now here's the point: A negaFibonacci code can be assigned to each cell of the pentagrid in such a way that the codes of its five neighbors are easy to compute. Let's call the neighbors n, s, e, w, and o, for "north," "south," "east," "west," and "other." If α is the code assigned to a given cell, we define

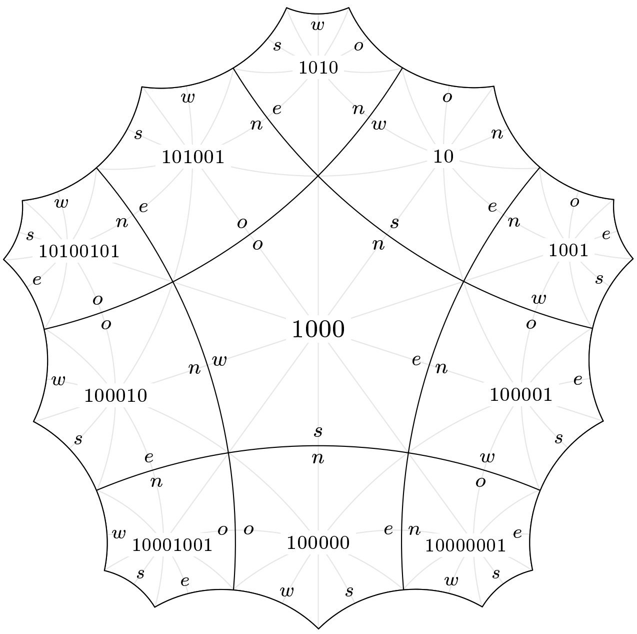
$$\alpha_n = \alpha \gg 2, \quad \alpha_s = \alpha \ll 2, \quad \alpha_e = \alpha_s{+}, \quad \alpha_w = \alpha_s{-}; \tag{150}$$

thus $\alpha_{sn} = \alpha$, and also $\alpha_{en} = (\alpha 01)_n = \alpha$. The "other" direction is trickier:

$$\alpha_o = \begin{cases} \alpha_n{+}, & \text{if } \alpha \,\&\, 1 = 1; \\ \alpha_w{-}, & \text{if } \alpha \,\&\, 1 = 0. \end{cases} \tag{151}$$

For example, $1000_o = 101001$ and $101001_o = 1000$. This mysterious interloper lies between north and east when α ends with 1, but between north and west when α ends with 0.

If we choose any cell and label it with code 0, and if we also choose an orientation so that its neighbors are n, e, s, w, and o in clockwise order, rules (150) and (151) will assign consistent labels to every cell of the pentagrid. (See exercise 160.) For example, the vicinity of a cell labeled 1000 will look like this:



(152)

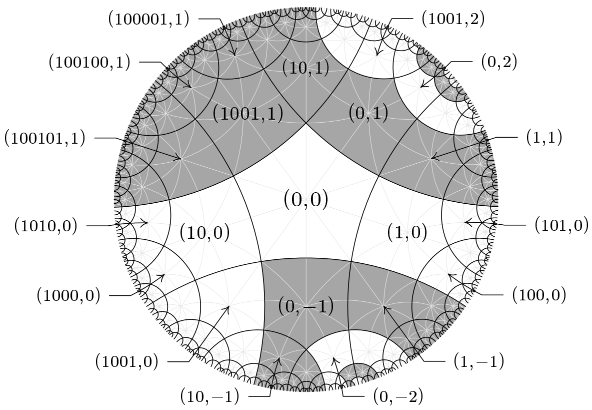
The code labels do not, however, identify cells uniquely, because infinitely many cells receive the same label. (Indeed, we clearly have $0_n = 0_s = 0$ and $1_w = 1_o = 1$.) To get a unique identifier, we attach a second coordinate so that each cell's full name has the form (α, y), where y is an integer. When y is constant and α ranges over all negaFibonacci codes, the cells (α, y) form a more-or-less hook-shaped strip whose edges take a 90° turn next to cell $(0, y)$. In general, the five neighbors of (α, y) are $(\alpha, y)_n = (\alpha_n, y + \delta_n(\alpha))$, $(\alpha, y)_s = (\alpha_s, y + \delta_s(\alpha))$,

$(\alpha, y)_e = (\alpha_e, y + \delta_e(\alpha))$, $(\alpha, y)_w = (\alpha_w, y + \delta_w(\alpha))$, and $(\alpha, y)_o = (\alpha_o, y + \delta_o(\alpha))$, where

$$\delta_n(\alpha) = [\alpha = 0], \quad \delta_s(\alpha) = -[\alpha = 0], \quad \delta_e(\alpha) = 0, \quad \delta_w(\alpha) = -[\alpha = 1];$$

$$\delta_o(\alpha) = \begin{cases} \operatorname{sign}(\alpha_o - \alpha_n)[\alpha_o \,\&\, \alpha_n = 0], & \text{if } \alpha \,\&\, 1 = 1; \\ \operatorname{sign}(\alpha_o - \alpha_w)[\alpha_o \,\&\, \alpha_w = 0], & \text{if } \alpha \,\&\, 1 = 0. \end{cases} \qquad (153)$$

(See the illustration below.) Bitwise operations now allow us to surf the entire hyperbolic plane with ease. On the other hand, we could also ignore the y coordinates as we move, thereby wrapping around a "hyperbolic cylinder" of pentagons; the α coordinates define an interesting multigraph on the set of all negaFibonacci codes, in which every vertex has degree 5.



(154)

Bitmap graphics. It's fun to write programs that deal with pictures and shapes, because they involve our left and right brains simultaneously. When image data is involved, the results can be engrossing even if there are bugs in our code.

The book you are now reading was typeset by software that treated each page as a gigantic matrix of 0s and 1s, called a "raster" or "bitmap," containing millions of square picture elements called "pixels." The rasters were transmitted to printing machines, causing tiny dots of ink to be placed wherever a 1 appeared in the matrix. Physical properties of ink and paper caused those small clusters of dots to look like smooth curves; but each pixel's basic squareness becomes evident if we enlarge the images tenfold, as in the letter 'A' shown in Fig. 15(a).

With bitwise operations we can achieve special effects like "custering," in which the black pixels disappear when they are surrounded on all sides:

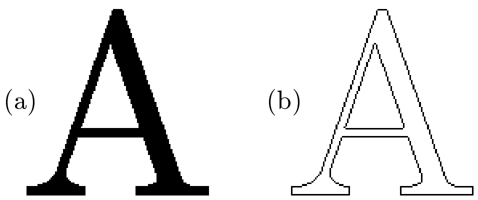


Fig. 15. The letter A, before and after custering.

cylinder
–implicit data structures
bitmap graphics–
typeset
raster
pixels
printing
custering

This operation, introduced by R. A. Kirsch, L. Cahn, C. Ray, and G. E. Urban [*Proc. Eastern Joint Computer Conf.* **12** (1957), 221–229], can be expressed as

$$\text{custer}(X) \;=\; X \mathbin{\&} {\sim}\big((X \mathbin{\vee\!\!\vee} 1) \mathbin{\&} (X \gg 1) \mathbin{\&} (X \ll 1) \mathbin{\&} (X \mathbin{\wedge\!\!\wedge} 1)\big), \tag{155}$$

where '$X \mathbin{\vee\!\!\vee} 1$' and $X \mathbin{\wedge\!\!\wedge} 1$' stand respectively for the result of shifting the bitmap X down or up by one row. Let us write

$$X_{\rm N} = X \mathbin{\vee\!\!\vee} 1, \quad X_{\rm W} = X \gg 1, \quad X_{\rm E} = X \ll 1, \quad X_{\rm S} = X \mathbin{\wedge\!\!\wedge} 1 \tag{156}$$

for the 1-pixel shifts of a bitmap X. Then, for example, the symbolic expression '$X_{\rm N} \mathbin{\&} (X_{\rm S} \mid \overline{X}_{\rm E})$' evaluates to 1 in those pixel positions whose northern neighbor is black, and which also have either a black neighbor on the south side or a white neighbor to the east. With these abbreviations, (155) takes the form

$$\text{custer}(X) \;=\; X \mathbin{\&} {\sim}(X_{\rm N} \mathbin{\&} X_{\rm W} \mathbin{\&} X_{\rm E} \mathbin{\&} X_{\rm S}), \tag{157}$$

which can also be expressed as $X \mathbin{\&} (\overline{X}_{\rm N} \mid \overline{X}_{\rm W} \mid \overline{X}_{\rm E} \mid \overline{X}_{\rm S})$.

Every pixel has four "rook-neighbors," with which it shares an edge at the top, left, right, or bottom. It also has eight "king-neighbors," with which it shares at least one corner point. For example, the king-neighbors that lie to the northeast of all pixels in a bitmap X can be denoted by $X_{\rm NE}$, which is equivalent to $(X_{\rm N})_{\rm E}$ in pixel algebra. Notice that we also have $X_{\rm NE} = (X_{\rm E})_{\rm N}$.

A 3×3 *cellular automaton* is an array of pixels that changes dynamically via a sequence of local transformations, all performed simultaneously: The state of each pixel at time $t+1$ depends entirely on its state at time t and the states of its king-neighbors at that time. Thus the automaton defines a sequence of bitmaps $X^{(0)}$, $X^{(1)}$, $X^{(2)}$, ... that lead from any given initial state $X^{(0)}$, where

$$X^{(t+1)} \;=\; f(X^{(t)}_{\rm NW}, X^{(t)}_{\rm N}, X^{(t)}_{\rm NE}, X^{(t)}_{\rm W}, X^{(t)}, X^{(t)}_{\rm E}, X^{(t)}_{\rm SW}, X^{(t)}_{\rm S}, X^{(t)}_{\rm SE}) \tag{158}$$

and f is any bitwise Boolean function of nine variables. Fascinating patterns often emerge in this way. For example, after Martin Gardner introduced John Conway's game of Life to the world in 1970, more computer time was probably devoted to studying its implications than to any other computational task during the next several years — although the people paying the computer bills were rarely told! (See exercise 167.)

There are 2^{512} Boolean functions of nine variables, so there are 2^{512} different 3×3 cellular automata. Many of them are trivial, but most of them probably have such complicated behavior that they are humanly impossible to understand. Fortunately there also are many cases that do turn out to be useful in practice — and much easier to justify on economic grounds than the simulation of a game.

For example, algorithms for recognizing alphabetic characters, fingerprints, or similar patterns often make use of a "thinning" process, which removes excess black pixels and reduces each component of the image to an underlying skeleton that is comparatively simple to analyze. Several authors have proposed cellular automata for this problem, beginning with D. Rutovitz [*J. Royal Stat. Society* **A129** (1966), 512–513] who suggested a 4×4 scheme. But parallel algorithms are notoriously subtle, and flaws tended to turn up after various methods had

Kirsch
Cahn
Ray
Urban
black
white
rook-neighbors
king-neighbors
8-neighbors, see king-neighbors
4-neighbors, see rook-neighbors
pixel algebra
cellular automaton
Gardner
Conway
Life
game
pattern recognition
optical character recognition
fingerprints
thinning
Rutovitz

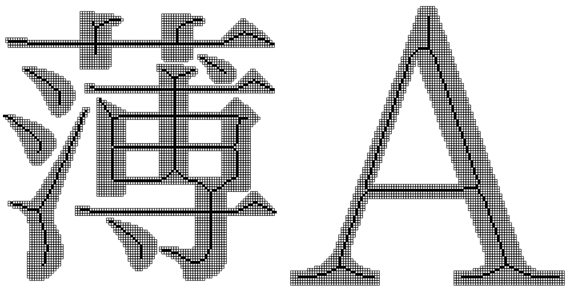


Fig. 16. Example results of Guo and Hall's 3×3 automaton for thinning the components of a bitmap. ("Hollow" pixels were originally black.)

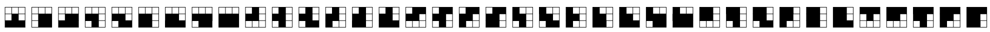
Guo
Hall
connectivity structure
kingwise connected
rookwise connected
Rosenfeld

been published. For example, at least two of the black pixels in a component like  should be removed, yet a symmetrical scheme will erroneously erase all four.

A satisfactory solution to the thinning problem was finally found by Z. Guo and R. W. Hall [*CACM* **32** (1989), 359–373, 759], using a 3×3 automaton that invokes alternate rules on odd and even steps. Consider the function

$$f(x_{\rm NW}, x_{\rm N}, x_{\rm NE}, x_{\rm W}, x, x_{\rm E}, x_{\rm SW}, x_{\rm S}, x_{\rm SE}) = x \wedge \neg g(x_{\rm NW}, \ldots, x_{\rm W}, x_{\rm E}, \ldots, x_{\rm SE}), \quad (159)$$

where $g = 1$ only in the following 37 configurations surrounding a black pixel:



Then we use (158), but with $f(x_{\rm NW}, x_{\rm N}, x_{\rm NE}, x_{\rm W}, x, x_{\rm E}, x_{\rm SW}, x_{\rm S}, x_{\rm SE})$ replaced by its 180° rotation $f(x_{\rm SE}, x_{\rm S}, x_{\rm SW}, x_{\rm E}, x, x_{\rm W}, x_{\rm NE}, x_{\rm N}, x_{\rm NW})$ on even-numbered steps. The process stops when two consecutive cycles make no change.

With this rule Guo and Hall proved that the 3×3 automaton will preserve the connectivity structure of the image, in a strong sense that we will discuss below. Furthermore their algorithm obviously leaves an image intact if it is already so thin that it contains no three pixels that are king-neighbors of each other. On the other hand it usually succeeds in "removing the meat off the bones" of each black component, as shown in Fig. 16. Slightly thinner thinning is obtained in certain cases if we add four additional configurations

 (160)

to the 37 listed above. In either case the function g can be evaluated with a Boolean chain of length 25. (See exercises 170–172.)

In general, the black pixels of an image can be grouped into segments or components that are *kingwise connected*, in the sense that any black pixel can be reached from any other pixel of its component by a sequence of king moves through black pixels. The white pixels also form components, which are *rookwise connected*: Any two white cells of a component are mutually reachable via rook moves that touch nothing black. It's best to use different kinds of connectedness for white and black, in order to preserve the topological concepts of "inside" and "outside" that are familiar from continuous geometry [see A. Rosenfeld, *JACM* **17** (1970), 146–160]. If we imagine that the corner points of a raster are black, an infinitely thin black curve can cross between pixels at a corner, but a white curve cannot. (We could also imagine white corner points, which would lead to rookwise connectivity for black and kingwise connectivity for white.)

Levialdi
Beyer
Cheshire cat
online
background
objects
holes
oriented tree

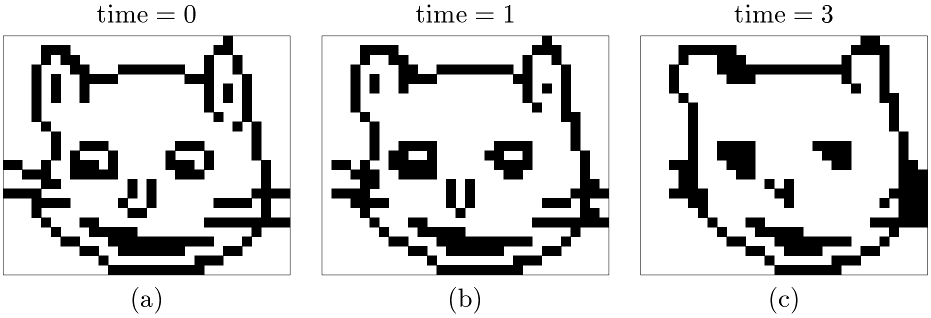


Fig. 17. The shrinking of a Cheshire cat

An amusing algorithm for shrinking a picture while preserving its connectivity, except that isolated black or white pixels disappear, was presented by S. Levialdi in *CACM* **15** (1972), 7–10; an equivalent algorithm, but with black and white reversed, had also appeared in T. Beyer's Ph.D. thesis (M.I.T., 1969). The idea is to use a cellular automaton with the simple transition function

$$f(x_{\rm NW}, x_{\rm N}, x_{\rm NE}, x_{\rm W}, x, x_{\rm E}, x_{\rm SW}, x_{\rm S}, x_{\rm SE}) = (x \wedge (x_{\rm W} \vee x_{\rm SW} \vee x_{\rm S})) \vee (x_{\rm W} \wedge x_{\rm S}) \quad (161)$$

at each step. This formula is actually a 2×2 rule, but we still need a 3×3 window if we want to keep track of the cases when a one-pixel component goes away.

For example, the 25×30 picture of a Cheshire cat in Fig. 17(a) has seven kingwise black components: the outline of its head, the two earholes, the two eyes, the nose, and the smile. The result after one application of (161) is shown in Fig. 17(b): Seven components remain, but there's an isolated point in one ear, and the other earhole will become isolated after the next step. Hence Fig. 17(c) has only five components. After six steps the cat loses its nose, and even the smile will be gone at time 14. Sadly, the last bit of cat will vanish during step 46.

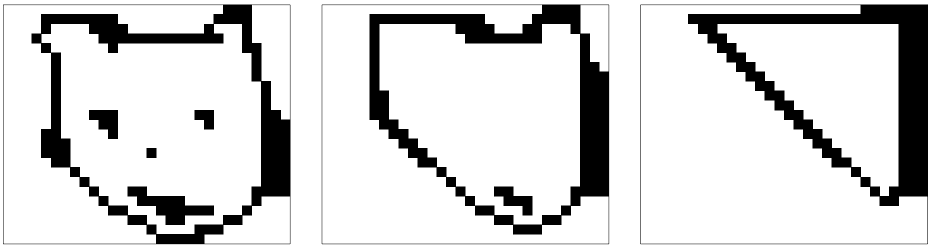
At most $M + N - 1$ transitions will wipe out any $M \times N$ picture, because the lowest visible northwest-to-southeast diagonal line moves relentlessly upward each time. Exercises 176 and 177 prove that different components will never merge together and interfere with each other.

Of course this cubic-time cellular method isn't the fastest way to count or identify the components of a picture. We can actually do that job "online," while looking at a large image one row at a time, not bothering to keep all of the previously seen rows in memory if we don't wish to look at them again.

While we're analyzing the components we might as well also record the relationships between them. Let's assume that only finitely many black pixels are present. Then there's an infinite component of white pixels called the *background*. Black components adjacent to the background constitute the main *objects* of the image. And these objects may in turn have *holes*, which may serve as a background for another level of objects, and so on. Thus the connected components of any finite picture form a hierarchy — an oriented tree, rooted at the background. Black components appear at the odd-numbered levels of this tree, and white components at the even-numbered levels, alternating between

surrounded
simply connected

time = 5 time = 10 time = 20

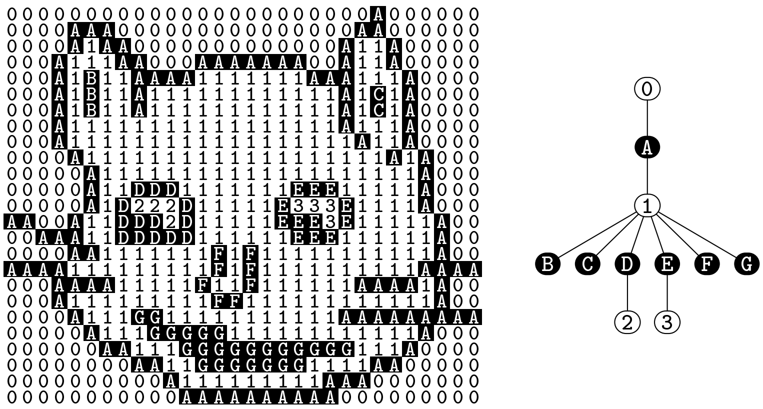


(d) (e) (f)

by repeated application of Levialdi's transformation.

kingwise and rookwise connectedness. Each component except the background is *surrounded* by its parent. Childless components are said to be *simply connected.*

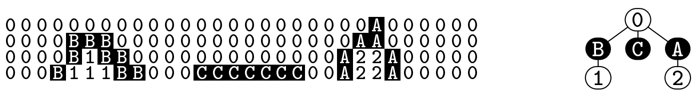
For example, here are the Cheshire cat's components, labeled with digits for white pixels and letters for the black ones, and the corresponding oriented tree:



(162)

During the shrinking process of Fig. 17, components disappear in the order 🅒, {🅑, ②, ③} (all at time 3), 🅕, 🅔, 🅓, 🅖, ①, 🅐.

Suppose we want to analyze the components of such a picture by reading one row at a time. After we've seen four rows the result-so-far will be



(163)

and we'll be ready to scan row five. A comparison of rows four and five will then show that 🅑 and 🅒 should merge into 🅐, but that new components 🅑 and ③ should also be launched. Exercise 179 contains full details about an instructive algorithm that properly updates the current tree as new rows are input. Additional information can also be computed on the fly: For example, we could determine the area of each component, the locations of its first and last pixels, the smallest enclosing rectangle, and/or its center of gravity.

raster graphics
conic sections
circles
ellipses
parabolas
hyperbolas
Cartesian coordinates
boundary curves++++
edge transitions++++
custerized
Ackland
Weste
Jordan curve theorem
Veblen
inside
outside
digitization

***Filling.** Let's complete our quick tour of raster graphics by considering how to fill regions that are bounded by straight lines and/or simple curves. Particularly efficient algorithms are available when the curves are built up from "conic sections" — circles, ellipses, parabolas, or hyperbolas, as in classical geometry.

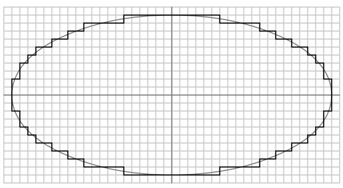
In keeping with geometric tradition, we shall adopt Cartesian coordinates (x, y) in the following discussion, instead of speaking about rows or columns of pixels: An increase of x will signify a move to the right, while an increase of y will move upward. More significantly, we will focus on the *edges* between square pixels, instead of on the pixels themselves. Edges run between integer points (x, y) and (x', y') of the plane when $|x - x'| + |y - y'| = 1$. Each pixel is bounded by the four edges (x, y) — $(x-1, y)$ — $(x-1, y-1)$ — $(x, y-1)$ — (x, y). Experience has shown that algorithms for filling contours become simpler and faster when we concentrate on the edge transitions between white and black, instead of on the black pixels of a custerized boundary. (See, for example, the discussion by B. D. Ackland and N. Weste in *IEEE Trans.* **C-30** (1981), 41–47.)

Consider a continuous curve $z(t) = \bigl(x(t), y(t)\bigr)$ that is traced out as t varies from 0 to 1. We assume that the curve doesn't intersect itself for $0 \le t < 1$, and that $z(0) = z(1)$. The famous Jordan curve theorem [C. Jordan, *Cours d'analyse* **3** (1887), 587–594; O. Veblen, *Trans. Amer. Math. Soc.* **6** (1905), 83–98] states that every such curve divides the plane into two regions, called the inside and the outside. We can "digitize" $z(t)$ by forcing it to travel along edges between pixels; then we obtain an approximation in which the inside pixels are black and the outside pixels are white. This digitization process essentially replaces the original curve by the sequence of integer points

$$\mathrm{round}(z(t)) \;=\; \bigl(\lfloor x(t) + \tfrac{1}{2}\rfloor, \lfloor y(t) + \tfrac{1}{2}\rfloor\bigr), \qquad \text{for } 0 \le t \le 1. \tag{164}$$

The curve can be perturbed slightly, if necessary, so that $z(t)$ never passes exactly through the center of a pixel. Then the digitized curve takes discrete steps along pixel edges as t grows; and a pixel lies inside the digitization if and only if its center lies inside the original continuous curve $\{z(t) \mid 0 \le t \le 1\}$.

For example, the equations $x(t) = 20\cos 2\pi t$ and $y(t) = 10 \sin 2\pi t$ define an ellipse. Its digitization, $\mathrm{round}(z(t))$, starts at $(20, 0)$ when $t = 0$, then jumps to $(20, 1)$ when $t \approx .008$ and $10\sin 2\pi t = 0.5$. Then it proceeds to the points $(20, 2)$, $(19, 2)$, $(19, 3)$, $(19, 4)$, $(18, 4)$, $\ldots$, $(20, -1)$, $(20, 0)$, as t increases through the values .024, .036, .040, .057, .062, $\ldots$, .976, .992:



(165)

The horizontal edges of such a boundary are conveniently represented by bit vectors $H(y)$ for each y; for example, $H(10) = \ldots 000000111111111111000000\ldots$ and $H(9) = \ldots 01111100000000000000111110\ldots$ in (165). If the ellipse is filled

quadratic form
Pitteway
three-register algorithm+++

with black to obtain a bitmap B, the H vectors mark transitions between black and white, so we have the symbolic relation

$$H \;=\; B \oplus (B \mathbin{\Lsh} 1). \tag{166}$$

Conversely, it's easy to obtain B when the H vectors are given:

$$\begin{aligned} B(y) &= H(y_{\max}) \oplus H(y_{\max-1}) \oplus \cdots \oplus H(y+1) \\ &= H(y_{\min}) \oplus H(y_{\min+1}) \oplus \cdots \oplus H(y). \end{aligned} \tag{167}$$

Notice that $H(y_{\min}) \oplus H(y_{\min+1}) \oplus \cdots \oplus H(y_{\max})$ is the zero vector, because each bitmap is white at both top and bottom. Notice further that the analogous *vertical* edge vectors $V(x)$ are redundant: They satisfy the formulas $V = B \oplus (B \ll 1)$ and $B = V^{\oplus}$ (see exercise 36), but we need not bother to keep track of them.

Conic sections are easier to deal with than most other curves, because we can readily eliminate the parameter t. For example, the ellipse that led to (165) can be defined by the equation $(x/20)^2 + (y/10)^2 = 1$, instead of using sines and cosines. Therefore pixel (x, y) should be black if and only if its center point $(x-\frac{1}{2}, y-\frac{1}{2})$ lies inside the ellipse, if and only if $(x-\frac{1}{2})^2/400 + (y-\frac{1}{2})^2/100 - 1 < 0$.

In general, every conic section is the set of points for which $F(x, y) = 0$, when F is an appropriate quadratic form. Therefore there's a quadratic form

$$Q(x, y) \;=\; F(x - \tfrac{1}{2}, y - \tfrac{1}{2}) \;=\; ax^2 + bxy + cy^2 + dx + ey + f \tag{168}$$

that is negative at the integer point (x, y) if and only if pixel (x, y) lies on a given side of the digitized curve.

For practical purposes we may assume that the coefficients $(a, b, \ldots, f)$ of Q are not-too-large integers. Then we're in luck, because the exact value of $Q(x, y)$ is easy to compute. In fact, as pointed out by M. L. V. Pitteway [*Comp. J.* **10** (1967), 282–289], there's a nice "three-register algorithm" by which we can quickly track the boundary points: Let x and y be integers, and suppose we've got the values of $Q(x, y)$, $Q_x(x, y)$, and $Q_y(x, y)$ in three registers (Q, Q_x, Q_y), where

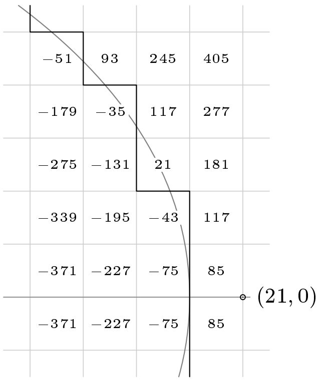
$$Q_x(x, y) = 2ax + by + d \qquad \text{and} \qquad Q_y(x, y) = bx + 2cy + e \tag{169}$$

are $\frac{\partial}{\partial x}Q$ and $\frac{\partial}{\partial y}Q$. We can then move to any adjacent integer point, because

$$\begin{aligned} &Q(x\pm1, y) = Q(x, y) \pm Q_x(x, y) + a, && Q(x, y\pm1) = Q(x, y) \pm Q_y(x, y) + c, \\ &Q_x(x\pm1, y) = Q_x(x, y) \pm 2a, && Q_x(x, y\pm1) = Q_x(x, y) \pm b, \\ &Q_y(x\pm1, y) = Q_y(x, y) \pm b; && Q_y(x, y\pm1) = Q_y(x, y) \pm 2c. \end{aligned} \tag{170}$$

Furthermore we can divide the contour into separate pieces, in each of which $x(t)$ and $y(t)$ are both monotonic. For example, when the ellipse (165) travels from $(20, 0)$ to $(0, 10)$, the value of x decreases while y increases; thus we need only move from (x, y) to $(x-1, y)$ or to $(x, y+1)$. If registers (Q, R, S) respectively hold $(Q, Q_x - a, Q_y + c)$, a move to $(x-1, y)$ simply sets $Q \leftarrow Q - R$, $R \leftarrow R - 2a$, and $S \leftarrow S - b$; a move to $(x, y+1)$ is just as quick. With care, this idea leads to a blindingly fast way to discover the correctly digitized edges of any conic curve.

For example, the quadratic form $Q(x,y)$ for ellipse (165) is $4x^2 + 16y^2 - (4x + 16y + 1595)$, when we integerize its coefficients. We have $Q(20,0) = F(19.5,-0.5) = -75$ and $Q(21,0) = +85$; therefore pixel $(20,0)$, whose center is $(19.5,-0.5)$, is inside the ellipse, but pixel $(21,0)$ isn't. Let's zoom in closer:



(171)

The boundary can be deduced without examining Q at very many points. In fact, we don't need to look at $Q(21,0)$, because we know that all edges between $(20,0)$ and $(0,10)$ must go either upwards or to the left. First we test $Q(20,1)$ and find it negative (-75); so we move up. Also $Q(20,2)$ is negative (-43), so we go up again. Then we test $Q(20,3)$, and find it positive (21); so we move left. And so on. Only the Q values -75, -43, 21, -131, -35, 93, -51, ... actually need to be examined, if we've set the three-register method up properly.

Algorithm T (*Three-register algorithm for conics*). Given two integer points (x,y) and (x',y'), and an integer quadratic form Q as in (168), this algorithm decides how to digitize a portion of the conic section defined by $F(x,y) = 0$, where $F(x,y) = Q(x+\frac{1}{2}, y+\frac{1}{2})$. It creates $|x'-x|$ horizontal edges and $|y'-y|$ vertical edges, which form a path from (x,y) to (x',y'). We assume that

i) Real-valued points (ξ,η) and (ξ',η') exist such that $F(\xi,\eta) = F(\xi',\eta') = 0$.

ii) The curve travels from (ξ,η) to (ξ',η') monotonically in both coordinates.

iii) $x = \lfloor \xi + \frac{1}{2} \rfloor$, $y = \lfloor \eta + \frac{1}{2} \rfloor$, $x' = \lfloor \xi' + \frac{1}{2} \rfloor$, and $y' = \lfloor \eta' + \frac{1}{2} \rfloor$.

iv) If we traverse the curve from (ξ,η) to (ξ',η'), we see $F < 0$ on our left.

T1. [Initialize.] If $x = x'$, go to T11; if $y = y'$, go to T10. If $x < x'$ and $y < y'$, set $Q \leftarrow Q(x{+}1, y{+}1)$, $R \leftarrow Q_x(x{+}1, y{+}1){+}a$, $S \leftarrow Q_y(x{+}1, y{+}1){+}c$, and go to T2. If $x < x'$ and $y > y'$, set $Q \leftarrow Q(x{+}1, y)$, $R \leftarrow Q_x(x{+}1, y) + a$, $S \leftarrow Q_y(x{+}1, y) - c$, and go to T3. If $x > x'$ and $y < y'$, set $Q \leftarrow Q(x, y{+}1)$, $R \leftarrow Q_x(x, y{+}1) - a$, $S \leftarrow Q_y(x, y{+}1) + c$, and go to T4. If $x > x'$ and $y > y'$, set $Q \leftarrow Q(x,y)$, $R \leftarrow Q_x(x,y) - a$, $S \leftarrow Q_y(x,y) - c$, and go to T5.

T2. [Right or up.] If $Q < 0$, do T9; otherwise do T6. Repeat until interrupted.

T3. [Down or right.] If $Q < 0$, do T7; otherwise do T9. Repeat until interrupted.

T4. [Up or left.] If $Q < 0$, do T6; otherwise do T8. Repeat until interrupted.

circles
tiling
hyperbolic plane
eofill

T5. [Left or down.] If $Q < 0$, do T8; otherwise do T7. Repeat until interrupted.

T6. [Move up.] Create the edge (x, y) — $(x, y{+}1)$, then set $y \leftarrow y{+}1$. Interrupt to T10 if $y = y'$; otherwise set $Q \leftarrow Q + S$, $R \leftarrow R + b$, $S \leftarrow S + 2c$.

T7. [Move down.] Create the edge (x, y) — $(x, y{-}1)$, then set $y \leftarrow y - 1$. Interrupt to T10 if $y = y'$; otherwise set $Q \leftarrow Q{-}S$, $R \leftarrow R{-}b$, $S \leftarrow S{-}2c$.

T8. [Move left.] Create the edge (x, y) — $(x{-}1, y)$, then set $x \leftarrow x - 1$. Interrupt to T11 if $x = x'$; otherwise set $Q \leftarrow Q{-}R$, $R \leftarrow R{-}2a$, $S \leftarrow S{-}b$.

T9. [Move right.] Create the edge (x, y) — $(x{+}1, y)$, then set $x \leftarrow x + 1$. Interrupt to T11 if $x = x'$; otherwise set $Q \leftarrow Q{+}R$, $R \leftarrow R{+}2a$, $S \leftarrow S{+}b$.

T10. [Finish horizontally.] While $x < x'$, create the edge (x, y) — $(x{+}1, y)$ and set $x \leftarrow x + 1$. While $x > x'$, create the edge (x, y) — $(x{-}1, y)$ and set $x \leftarrow x - 1$. Terminate the algorithm.

T11. [Finish vertically.] While $y < y'$, create the edge (x, y) — $(x, y{+}1)$ and set $y \leftarrow y + 1$. While $y > y'$, create the edge (x, y) — $(x, y{-}1)$ and set $y \leftarrow y - 1$. Terminate the algorithm. ▌

For example, when this algorithm is invoked with $(x, y) = (20, 0)$, $(x', y') = (0, 10)$, and $Q(x, y) = 4x^2 + 16y^2 - 4x - 16y - 1595$, it will create the edges $(20, 0)$ — $(20, 1)$ — $(20, 2)$ — $(19, 2)$ — $(19, 3)$ — $(19, 4)$ — $(18, 4)$ — $(18, 5)$ — $(17, 5)$ — $(17, 6)$ — $\cdots$ — $(5, 9)$ — $(5, 10)$, then make a beeline for $(0, 10)$. (See (165) and (171).) Exercise 182 explains why it works.

Movement to the right in step T9 is conveniently implemented by setting $H(y) \leftarrow H(y) \oplus (1 \ll (x_{\max} - x))$, using the H vectors of (166) and (167). Movement to the left is similar, but we set $x \leftarrow x - 1$ first. Step T10 could set

$$H(y) \leftarrow H(y) \oplus ((1 \ll (x_{\max} + 1 - \min(x, x'))) - (1 \ll (x_{\max} - \max(x, x')))); \quad (172)$$

but one move at a time might be just as good, because $|x' - x|$ is often small. Movement up or down needs no action, because vertical edges are redundant.

Notice that the algorithm runs somewhat faster in the special case when $b = 0$; circles always belong to this case. The even more special case of straight lines, when $a = b = c = 0$, is of course faster yet; then we have a simple *one-register* algorithm (see exercise 184).

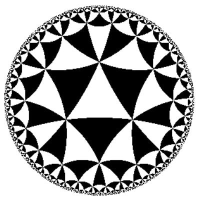


Fig. 18. Pixels change from white to black and back again, at the edges of digitized circles.

When many contours are filled in the same image, using H vectors, the pixel values change between black and white whenever we cross an odd number of edges. Figure 18 illustrates a tiling of the hyperbolic plane by equilateral 45°-45°-45° triangles, obtained by superimposing the results of several hundred applications of Algorithm T.

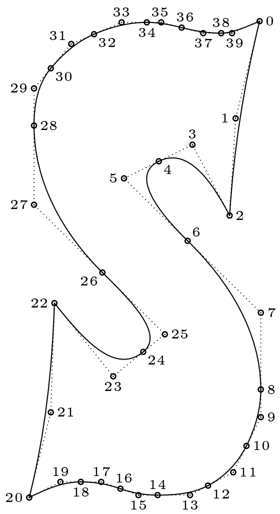


Fig. 19. Squines that define the outline contour of an ‘**S**’.

Bézier splines
squines
control points
S, the letter
drawing on a bitmap
–filling a contour in a bitmap
Hobby
–bitmap graphics
conditional
branch
CSNZ
pipelined machine

Algorithm T applies only to conic curves. But that's not really a limitation in practice, because just about every shape we ever need to draw can be well approximated by "piecewise conics" called quadratic Bézier splines or *squines.* For example, Fig. 19 shows a typical squine curve with 40 points $(z_0, z_1, \ldots, z_{39}, z_{40})$, where $z_{40} = z_0$. The even-numbered points $(z_0, z_2, \ldots, z_{40})$ lie on the curve; the others, $(z_1, z_3, \ldots, z_{39})$, are called "control points," because they regulate local bending and flexing. Each section $S(z_{2j}, z_{2j+1}, z_{2j+2})$ begins at point z_{2j}, traveling in direction $z_{2j+1} - z_{2j}$. It ends at point z_{2j+2}, traveling in direction $z_{2j+2} - z_{2j+1}$. Thus if z_{2j} lies on the straight line from z_{2j-1} to z_{2j+1}, the squine passes smoothly through point z_{2j} without changing direction.

Exercise 185 defines $S(z_{2j}, z_{2j+1}, z_{2j+2})$ precisely, and exercise 186 explains how to digitize any squine curve using Algorithm T. The region inside the digitized edges can then be filled with black pixels.

Incidentally, the task of *drawing* lines and curves on a bitmap turns out to be much more difficult than the task of *filling* a digitized contour, because we want diagonal strokes to have the same apparent thickness as vertical and horizontal strokes do. An excellent solution to the line-drawing problem was found by John D. Hobby, *JACM* **36** (1989), 209–229.

***Branchless computation.** Modern computers tend to slow down when a program contains conditional branch instructions, because an uncertain flow of control can interfere with predictive lookahead circuitry. Therefore we've used `MMIX`'s conditional-set instructions like `CSNZ` in programs like (56). Indeed, the four instructions '`SRU z,y,16; ADD t,lam,16; CSNZ y,q,z; CSNZ lam,q,t`' found in (56) are probably faster than their three-instruction counterpart

```
BZ q,@+12; SRU y,y,16; ADD lam,lam,16                    (173)
```

when the actual running time is measured on a highly pipelined machine, even though the rule-of-thumb cost of (173) is only $3v$ according to Table 1.3.1′–1.

Bitwise operations can help diminish the need for costly branching. For example, if MMIX didn't have a CSNZ instruction we could write

```
NEG m,q;   OR m,m,q;   SR m,m,63;
SRU t,y,16;   XOR t,t,y;   AND t,t,m;   XOR y,y,t;           (174)
ADD t,lam,16;   XOR t,t,lam;   AND t,t,m;   XOR lam,lam,t;
```

here the first line creates the mask $\mathtt{m} = -[q \neq 0]$. On some computers these eleven branchless instructions would still run faster than the three instructions in (173).

The inner loop of a merge sort algorithm provides an instructive example. Suppose we want to do the following operations repeatedly:

If $x_i < y_j$, set $z_k \leftarrow x_i$, $i \leftarrow i+1$, and go to *x_done* if $i = i_{\max}$.
Otherwise set $z_k \leftarrow y_j$, $j \leftarrow j+1$, and go to *y_done* if $j = j_{\max}$.
Then set $k \leftarrow k+1$ and go to *z_done* if $k = k_{\max}$.

If we implement them in the "obvious" way, four conditional branches are involved, three of which are active on each path through the loop:

```
1H CMP  t,xi,yj; BNN t,2F    Branch if x_i ≥ y_j.
   STO  xi,zbase,kk          z_k ← x_i.
   ADD  ii,ii,8              i ← i + 1.
   BZ   ii,X_Done            To x_done if i = i_max.
   LDO  xi,xbase,ii          Load x_i into register xi.
   JMP  3F                   Join the other branch.
2H STO  yj,zbase,kk          z_k ← y_j.
   ADD  jj,jj,8              j ← j + 1.
   BZ   jj,Y_Done            To y_done if j = j_max.
   LDO  yj,ybase,jj          Load y_j into register yj.
3H ADD  kk,kk,8              k ← k + 1.
   PBNZ kk,1B                Repeat if k ≠ k_max.
   JMP  Z_Done               To z_done. ▮
```

(Here $\mathtt{ii} = 8(i - i_{\max})$, $\mathtt{jj} = 8(j - j_{\max})$, and $\mathtt{kk} = 8(k - k_{\max})$; the factor of 8 is needed because x_i, y_j, and z_k are octabytes.) Those four branches can be reduced to just one:

```
1H CMP  t,xi,yj              t ← sign(x_i − y_j).
   CSN  yj,t,xi              yj ← min(x_i, y_j).
   STO  yj,zbase,kk          z_k ← yj.
   AND  t,t,8                t ← 8[x_i < y_j].
   ADD  ii,ii,t              i ← i + [x_i < y_j].
   LDO  xi,xbase,ii          Load x_i into register xi.
   XOR  t,t,8                t ← t ⊕ 8.
   ADD  jj,jj,t              j ← j + [x_i ≥ y_j].
   LDO  yj,ybase,jj          Load y_j into register yj.
   ADD  kk,kk,8              k ← k + 1.
   AND  u,ii,jj; AND u,u,kk  u ← ii & jj & kk.
   PBN  u,1B                 Repeat if i < i_max, j < j_max, and k < k_max. ▮
```

When the loop stops in this version, we can readily decide whether to continue at *x_done*, *y_done*, or *z_done*. These instructions load both x_i and y_j from memory each time, but the redundant value will already be present in the cache.

mask
signed shift right
merge sort
cache

***More applications of MOR and MXOR.** Let's finish off our study of bitwise manipulation by taking a look at two operations that are specifically designed for 64-bit work. `MMIX`'s instructions `MOR` and `MXOR`, which essentially carry out matrix multiplication on 8×8 Boolean matrices, turn out to be extremely flexible and powerful, both by themselves and in combination with other bitwise operations.

MOR++
MXOR++
matrix multiplication
mask
bit permutations
byte permutations
permutation matrix
inverse permutation
perfect shuffle
zippered
MUX
δ-swaps
finite field

If $x = (x_7 \ldots x_1 x_0)_{256}$ is an octabyte and $a = (a_7 \ldots a_1 a_0)_2$ is a single byte, the instruction `MOR t,x,a` sets $t \leftarrow a_7 x_7 \mid \cdots \mid a_1 x_1 \mid a_0 x_0$, while `MXOR t,x,a` sets $t \leftarrow a_7 x_7 \oplus \cdots \oplus a_1 x_1 \oplus a_0 x_0$. For example, `MOR t,x,2` and `MXOR t,x,2` both set $t \leftarrow x_1$; `MOR t,x,3` sets $t \leftarrow x_1 \mid x_0$; and `MXOR t,x,3` sets $t \leftarrow x_1 \oplus x_0$.

In general, of course, `MOR` and `MXOR` are functions of octabytes. When $y = (y_7 \ldots y_1 y_0)_{256}$ is a general octabyte, the instruction `MOR t,x,y` produces the octabyte t whose jth byte t_j is the result of `MOR` applied to x and y_j.

Suppose $x = -1 =$ `#ffffffffffffffff`. Then `MOR t,x,y` computes the mask t in which byte t_j is `#ff` whenever $y_j \neq 0$, while t_j is zero when $y_j = 0$. This simple special case is quite useful, because it accomplishes in just one instruction what we previously needed seven operations to achieve in situations like (92).

We observed in (66) that two `MOR`s will suffice to reverse the bits of any 64-bit word, and many other important bit permutations also become easy when `MOR` is in a computer's repertoire. Suppose π is a permutation of $\{0, 1, \ldots, 7\}$ that takes $0 \mapsto 0\pi$, $1 \mapsto 1\pi$, $\ldots$, $7 \mapsto 7\pi$. Then the octabyte $p = (2^{7\pi} \ldots 2^{1\pi} 2^{0\pi})_{256}$ corresponds to a permutation matrix that makes `MOR` do nice tricks: `MOR t,x,p` will *permute the bytes* of x, setting $t_j \leftarrow x_{j\pi}$. Furthermore, `MOR u,p,y` will *permute the bits* of each byte of y, according to the *inverse* permutation; it sets $u_j \leftarrow (a_7 \ldots a_1 a_0)_2$ when $y_j = (a_{7\pi} \ldots a_{1\pi} a_{0\pi})_2$.

With a little more skullduggery we can also expedite further permutations such as the perfect shuffle (76), which transforms a given octabyte $z = 2^{32}x + y = (x_{31} \ldots x_1 x_0 y_{31} \ldots y_1 y_0)_2$ into the "zippered" octabyte

$$w \;=\; x \ddagger y \;=\; (x_{31} y_{31} \ldots x_1 y_1 x_0 y_0)_2. \qquad (175)$$

With appropriate permutation matrices p, q, and r, the intermediate results

$$t = (x_{31}x_{27}x_{30}x_{26}x_{29}x_{25}x_{28}x_{24}y_{31}y_{27}y_{30}y_{26}y_{29}y_{25}y_{28}y_{24} \ldots \\ x_7x_3x_6x_2x_5x_1x_4x_0y_7y_3y_6y_2y_5y_1y_4y_0)_2, \qquad (176)$$

$$u = (y_{27}y_{31}y_{26}y_{30}y_{25}y_{29}y_{24}y_{28}x_{27}x_{31}x_{26}x_{30}x_{25}x_{29}x_{24}x_{28} \ldots \\ y_3y_7y_2y_6y_1y_5y_0y_4x_3x_7x_2x_6x_1x_5x_0x_4)_2 \qquad (177)$$

can be computed quickly via the four instructions

```
MOR t,z,p;  MOR t,q,t;  MOR u,t,r;  MOR u,r,u;        (178)
```

see exercise 203. So there's a mask m for which '`PUT rM,m; MUX w,t,u`' completes the perfect shuffle in just six cycles altogether. By contrast, the traditional method in exercise 53 requires 30 cycles (five δ-swaps).

The analogous instruction `MXOR` is especially useful when binary linear algebra is involved. For example, exercise 1.3.1′–37 shows that `XOR` and `MXOR` directly implement addition and multiplication in a finite field of 2^k elements, for $k \leq 8$.

The problem of *cyclic redundancy checking* provides an instructive example of another case where `MXOR` shines. Streams of data are often accompanied by "CRC bytes" in order to detect common types of transmission errors [see W. W. Peterson and D. T. Brown, *Proc. IRE* **49** (1961), 228–235]. One popular method, used for example in MP3 audio files, is to regard each byte $\alpha = (a_7 \ldots a_1 a_0)_2$ as if it were the polynomial

$$\alpha(x) \;=\; (a_7 \ldots a_1 a_0)_x \;=\; a_7 x^7 + \cdots + a_1 x + a_0. \qquad (179)$$

When transmitting n bytes $\alpha_{n-1} \ldots \alpha_1 \alpha_0$, we then compute the remainder

$$\beta = \bigl(\alpha_{n-1}(x) x^{8(n-1)} + \cdots + \alpha_1(x) x^8 + \alpha_0(x)\bigr) x^{16} \bmod p(x), \qquad (180)$$

where $p(x) = x^{16} + x^{15} + x^2 + 1$, using polynomial arithmetic mod 2, and append the coefficients of β as a 16-bit redundancy check.

The usual way to compute β is to process one byte at a time, according to classical methods like Algorithm 4.6.1D. The basic idea is to define the partial result $\beta_m = \bigl(\alpha_{n-1}(x) x^{8(n-1)} + \cdots + \alpha_m(x) x^{8m}\bigr) x^{16} \bmod p(x)$ so that $\beta_n = 0$, and then to use the recursion

$$\beta_m = ((\beta_{m+1} \ll 8) \mathbin{\&} {}^{\#}\mathtt{ff00}) \oplus \mathit{crc_table}[(\beta_{m+1} \gg 8) \oplus \alpha_m] \qquad (181)$$

to decrease m by 1 until $m = 0$. Here $\mathit{crc_table}[\alpha]$ is a 16-bit table entry that holds the remainder of $\alpha(x) x^{16}$, modulo $p(x)$ and mod 2, for $0 \le \alpha < 256$. [See A. Perez, *IEEE Micro* **3**, 3 (June 1983), 40–50.]

But of course we'd prefer to process 64 bits at once instead of 8. The solution is to find 8×8 matrices A and B such that

$$\alpha(x) x^{64} \equiv (\alpha A)(x) + (\alpha B)(x) x^{-8} \quad (\text{modulo } p(x) \text{ and } 2), \qquad (182)$$

for arbitrary bytes α, considering α to be a 1×8 vector of bits. Then we can pad the given data bytes $\alpha_{n-1} \ldots \alpha_1 \alpha_0$ with leading zeros so that n is a multiple of 8, and use the following efficient reduction method:

Begin with $c \leftarrow 0$, $n \leftarrow n - 8$, and $t \leftarrow (\alpha_{n+7} \ldots \alpha_n)_{256}$.
While $n > 0$, set $u \leftarrow t \cdot A$, $v \leftarrow t \cdot B$, $n \leftarrow n - 8$, (183)
$t \leftarrow (\alpha_{n+7} \ldots \alpha_n)_{256} \oplus u \oplus (v \gg 8) \oplus (c \ll 56)$, and $c \leftarrow v \mathbin{\&} {}^{\#}\mathtt{ff}$.

Here $t \cdot A$ and $t \cdot B$ denote matrix multiplication via `MXOR`. The desired CRC bytes, $(tx^{16} + cx^8) \bmod p(x)$, are then readily obtained from the 64-bit quantity t and the 8-bit quantity c. Exercise 210 contains full details; the total running time for n bytes comes to only $(\mu + 10v)n/8 + O(1)$.

The exercises below contain many more instances where `MOR` and `MXOR` lead to substantial economies. New tricks undoubtedly remain to be discovered.

For further reading. The book *Hacker's Delight* by Henry S. Warren, Jr. (Addison–Wesley, 2002) discusses bitwise operations in depth, emphasizing the great variety of options that are available on real-world computers that are not as ideal as `MMIX`.

cyclic redundancy checking
CRC
Peterson
Brown
MP3 (MPEG-1 Audio Layer III)
Perez
Warren

EXERCISES

▶ **1.** [*15*] What is the net effect of setting $x \leftarrow x \oplus y$, $y \leftarrow y \oplus (x \mathbin{\&} m)$, $x \leftarrow x \oplus y$?

2. [*16*] (H. S. Warren, Jr.) Are any of the following relations valid for all integers x and y? (i) $x \oplus y \le x \mid y$; (ii) $x \mathbin{\&} y \le x \mid y$; (iii) $|x - y| \le x \oplus y$.

3. [*M20*] If $x = (x_{n-1} \ldots x_1 x_0)_2$ with $x_{n-1} = 1$, let $x^M = (\bar{x}_{n-1} \ldots \bar{x}_1 \bar{x}_0)_2$. Thus we have $0^M, 1^M, 2^M, 3^M, \ldots = -1, 0, 1, 0, 3, 2, 1, 0, 7, 6, \ldots$, if we let $0^M = -1$. Prove that $(x \oplus y)^M < |x - y| \le x \oplus y$ for all $x, y \ge 0$.

▶ **4.** [*M16*] Let $x^C = \bar{x}$, $x^N = -x$, $x^S = x+1$, and $x^P = x-1$ denote the complement, the negative, the successor, and the predecessor of an infinite-precision integer x. Then we have $x^{CC} = x^{NN} = x^{SP} = x^{PS} = x$. What are x^{CN} and x^{NC}?

5. [*M21*] Prove or disprove the following conjectured laws concerning binary shifts:
a) $(x \ll j) \ll k = x \ll (j + k)$;
b) $(x \gg j) \mathbin{\&} (y \ll k) = ((x \gg (j + k)) \mathbin{\&} y) \ll k = (x \mathbin{\&} (y \ll (j + k))) \gg j$.

6. [*M22*] Find all integers x and y such that (a) $x \gg y = y \gg x$; (b) $x \ll y = y \ll x$.

7. [*M22*] (R. Schroeppel, 1972.) Find a fast way to convert the binary number $x = (\ldots x_2 x_1 x_0)_2$ to its negabinary counterpart $x = (\ldots x'_2 x'_1 x'_0)_{-2}$, and vice versa. *Hint:* Only two bitwise operations are needed!

▶ **8.** [*M22*] Given a finite set S of nonnegative integers, the "minimal excludant" of S is defined to be

$$\operatorname{mex}(S) = \min\{\, k \mid k \ge 0 \text{ and } k \notin S \,\}.$$

Let $x \oplus S$ denote the set $\{x \oplus y \mid y \in S\}$. Prove that if $x = \operatorname{mex}(S)$ and $y = \operatorname{mex}(T)$ then $x \oplus y = \operatorname{mex}((S \oplus y) \cup (x \oplus T))$.

9. [*M26*] (*Nim.*) Two people play a game with k piles of sticks, where there are a_j sticks in pile j. If $a_1 = \cdots = a_k = 0$ when it is a player's turn to move, that player loses; otherwise the player reduces one of the piles by any desired amount, throwing away the removed sticks, and it is the other player's turn. Prove that the player to move can force a victory if and only if $a_1 \oplus \cdots \oplus a_k \ne 0$.

10. [*HM40*] (*Conway's field.*) Continuing exercise 8, define the operation $x \otimes y$ of "nim multiplication" recursively by the formula

$$x \otimes y = \operatorname{mex}\bigl\{(x \otimes j) \oplus (i \otimes y) \oplus (i \otimes j) \bigm| 0 \le i < x,\ 0 \le j < y\bigr\}.$$

Prove that $\oplus$ and $\otimes$ define a *field* over the set of all nonnegative integers. Prove also that if $0 \le x, y < 2^{2^n}$ then $x \otimes y < 2^{2^n}$, and $x \otimes 2^{2^n} = 2^{2^n}x$. (In particular, this field contains subfields of size 2^{2^n} for all $n \ge 0$.) Explain how to compute $x \otimes y$ efficiently.

▶ **11.** [*M26*] (H. W. Lenstra, 1978.) Find a simple way to characterize all pairs of positive integers (m, n) for which $m \otimes n = mn$ in Conway's field.

12. [*M26*] Devise an algorithm for *division* in Conway's field. *Hint:* If $x < 2^{2^{n+1}}$ then we have $x \otimes (x \oplus (x \gg 2^n)) < 2^{2^n}$.

13. [*M32*] (*Second-order nim.*) Extend the game of exercise 9 by allowing two kinds of moves: Either a_j is reduced for some j, as before; or a_j is reduced and a_i is replaced by an arbitrary nonnegative integer, for some $i < j$. Prove that the player to move can now force a victory if and only if the pile sizes satisfy either $a_2 \ne a_3 \oplus \cdots \oplus a_k$ or $a_1 \ne a_3 \oplus (2 \otimes a_4) \oplus \cdots \oplus ((k-2) \otimes a_k)$. For example, when $k = 4$ and $(a_1, a_2, a_3, a_4) = (7, 5, 0, 5)$, the only winning move is to $(7, 5, 6, 3)$.

Warren
subtraction
complement
negative
infinite-precision
Schroeppel
negabinary
radix -2
minimal excludant
mex
Nim
game
Conway's field
nim multiplication
recursively
field
Lenstra
nim division
Nim, second-order

14. [*M30*] Suppose each node of a complete, infinite binary tree has been labeled with 0 or 1. Such a labeling is conveniently represented as a set $T = \{t, t_0, t_1, t_{00}, t_{01}, t_{10}, t_{11}, t_{000}, \ldots\}$, with one bit t_α for every binary string α; the root is labeled t, the left subtree labels are $T_0 = \{t_0, t_{00}, t_{01}, t_{000}, \ldots\}$, and the right subtree labels are $T_1 = \{t_1, t_{10}, t_{11}, t_{100}, \ldots\}$. Any such labeling can be used to transform a 2-adic integer $x = (\ldots x_2 x_1 x_0)_2$ into the 2-adic integer $y = (\ldots y_2 y_1 y_0)_2 = T(x)$ by setting $y_0 = t$, $y_1 = t_{x_0}$, $y_2 = t_{x_0 x_1}$, etc., so that $T(x) = 2T_{x_0}(\lfloor x/2 \rfloor) + t$. (In other words, x defines an infinite path in the binary tree, and y corresponds to the labels on that path, from right to left in the bit strings as we proceed from top to bottom of the tree.)

A *branching function* is the mapping $x^T = x \oplus T(x)$ defined by such a labeling. For example, if $t_{01} = 1$ and all of the other t_α are 0, we have $x^T = x \oplus 4[x \bmod 4 = 2]$.

a) Prove that every branching function is a permutation of the 2-adic integers.
b) For which integers k is $x \oplus (x \ll k)$ a branching function?
c) Let $x \mapsto x^T$ be a mapping from 2-adic integers into 2-adic integers. Prove that x^T is a branching function if and only if $\rho(x \oplus y) = \rho(x^T \oplus y^T)$ for all 2-adic x and y.
d) Prove that compositions and inverses of branching functions are branching functions. (Thus the set $\mathcal{B}$ of all branching functions is a permutation group.)
e) A branching function is *balanced* if the labels satisfy $t_\alpha = t_{\alpha 0} \oplus t_{\alpha 1}$ for all α. Show that the set of all balanced branching functions is a subgroup of $\mathcal{B}$.

▸ **15.** [*M21*] J. H. Quick noticed that $((x+2) \oplus 3) - 2 = ((x-2) \oplus 3) + 2$ for all x. Find all constants a and b such that $((x+a) \oplus b) - a = ((x-a) \oplus b) + a$ is an identity.

16. [*M31*] A function of x is called *animating* if it can be written in the form

$$((\ldots((((x + a_1) \oplus b_1) + a_2) \oplus b_2) + \cdots) + a_m) \oplus b_m$$

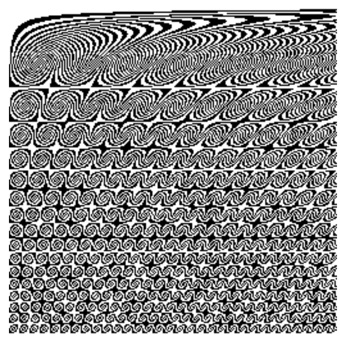
for some integer constants $a_1, b_1, a_2, b_2, \ldots, a_m, b_m$, with $m > 0$.

a) Prove that every animating function is a branching function (see exercise 14).
b) Furthermore, it is balanced if and only if $b_1 \oplus b_2 \oplus \cdots \oplus b_m = 0$. *Hint:* What binary tree labeling corresponds to the animating function $((x \oplus c) - 1) \oplus c$?
c) Let $\lfloor x \rceil = x \oplus (x-1) = 2^{\rho(x)+1} - 1$. Show that every balanced animating function can be written in the form

$$x \oplus \lfloor x \oplus p_1 \rceil \oplus \lfloor x \oplus p_2 \rceil \oplus \cdots \oplus \lfloor x \oplus p_l \rceil, \qquad p_1 < p_2 < \cdots < p_l,$$

for some integers $\{p_1, p_2, \ldots, p_l\}$, where $l \geq 0$, and this representation is unique.
d) Conversely, show that every such expression defines a balanced animating function.

17. [*HM36*] The results of exercise 16 make it possible to decide whether or not any two given animating functions are equal. Is there an algorithm that decides whether *any* given expression is identically zero, when that expression is constructed from a finite number of integer variables and constants using only the binary operations $+$ and $\oplus$? What if we also allow &?



18. [*M25*] The curious pixel pattern shown here has $(x^2 y \gg 11)$ & 1 in row x and column y, for $1 \leq x, y \leq 256$. Is there any simple way to explain some of its major characteristics mathematically?

complete, infinite binary tree
2-adic integer
branching function
permutation
ruler function rho
group
composition of permutations
balanced
Quick
XOR identities
animating
pixel pattern

▸ **19.** [*M37*] (*Paley's rearrangement theorem.*) Given three vectors $A = (a_0, \ldots, a_{2^n-1})$, $B = (b_0, \ldots, b_{2^n-1})$, and $C = (c_0, \ldots, c_{2^n-1})$ of nonnegative numbers, let

$$f(A,B,C) \;=\; \sum_{j\oplus k\oplus l=0} a_j b_k c_l.$$

For example, if $n = 2$ we have $f(A,B,C) = a_0b_0c_0 + a_0b_1c_1 + a_0b_2c_2 + a_0b_3c_3 + a_1b_0c_1 + a_1b_1c_0 + a_1b_2c_3 + \cdots + a_3b_3c_0$; in general there are 2^{2n} terms, one for each choice of j and k. Our goal is to prove that $f(A,B,C) \le f(A^*,B^*,C^*)$, where A^* denotes the vector A sorted into nonincreasing order, $a_0 \ge a_1 \ge \cdots \ge a_{2^n-1}$.

a) Prove the result when all elements of A, B, and C are 0s and 1s.
b) Show that it is therefore true in general.
c) Similarly, $f(A,B,C,D) = \sum_{j\oplus k\oplus l\oplus m=0} a_j b_k c_l d_m \le f(A^*,B^*,C^*,D^*)$.

▸ **20.** [*21*] (*Gosper's hack.*) The following seven operations produce a useful function y of x, when x is a positive integer. Explain what this function is and why it is useful.

$$u \leftarrow x \,\&\, {-x}; \qquad v \leftarrow x + u; \qquad y \leftarrow v + (((v \oplus x)/u) \gg 2).$$

21. [*22*] Construct the *reverse* of Gosper's hack: Show how to compute x from y.

22. [*21*] Implement Gosper's hack efficiently with MMIX code, assuming that $x < 2^{64}$, without using division.

▸ **23.** [*27*] A sequence of nested parentheses can be represented as a binary number by putting a 1 in the position of each right parenthesis. For example, '(())()' corresponds in this way to $(001101)_2$, the number 13. Call such a number a *parenthesis trace.*

a) What are the smallest and largest parenthesis traces that have exactly m 1s?
b) Suppose x is a parenthesis trace and y is the next larger parenthesis trace with the same number of 1s. Show that y can be computed from x with a short chain of operations analogous to Gosper's hack.
c) Implement your method on MMIX, assuming that $\nu x \le 32$.

▸ **24.** [*M30*] Program 1.3.2′P instructed MMIX to produce a table of the first five hundred prime numbers, using trial division to establish primality. Write an MMIX program that uses the "sieve of Eratosthenes" (exercise 4.5.4–8) to build a table of all odd primes that are less than N, packed into octabytes $Q_0, Q_1, \ldots, Q_{N/128}$ as in (27). Assume that $N \le 2^{32}$, and that it's a multiple of 128. What is the running time when $N = 3584$?

▸ **25.** [*15*] Four volumes sit side by side on a bookshelf. Each of them contains exactly 500 pages, printed on 250 sheets of paper 0.1 mm thick; each book also has a front and back cover whose thicknesses are 1 mm each. A bookworm gnaws its way from page 1 of Volume 1 to page 500 of Volume 4. How far does it travel while doing so?

26. [*22*] Suppose we want random access to a table of 12 million items of 5-bit data. We could pack 12 such items into one 64-bit word, thereby fitting the table into 8 megabytes of memory. But random access then seems to require division by 12, which is rather slow; we might therefore prefer to let each item occupy a full byte, thus using 12 megabytes altogether.

Show, however, that there's a memory-efficient approach that avoids division.

27. [*21*] In the notation of Eqs. (32)–(43), how would you compute (a) $(\alpha 10^a 01^b)_2$? (b) $(\alpha 10^a 11^b)_2$? (c) $(\alpha 00^a 01^b)_2$? (d) $(0^\infty 11^a 00^b)_2$? (e) $(0^\infty 01^a 00^b)_2$? (f) $(0^\infty 11^a 11^b)_2$?

28. [*16*] What does the operation $(x{+}1) \,\&\, \bar{x}$ produce?

29. [*20*] (V. R. Pratt.) Express the magic mask μ_k of (47) in terms of μ_{k+1}.

Paley
sorted
zero-one principle
0–1 principle
Gosper's hack
nested parentheses
parenthesis trace
Gosper's hack
MMIX
prime numbers
sieve
Eratosthenes
bookworm
pack
allocation of memory
storage allocation
division, avoiding
Pratt
magic mask

30. [*20*] If $x = 0$, the MMIX instructions (46) will set $\rho \leftarrow 64$ (which is a close enough approximation to ∞). What changes to (50) and (51) will produce the same result?

▶ **31.** [*20*] A mathematician named Dr. L. I. Presume decided to calculate the ruler function with a simple loop as follows: "Set $\rho \leftarrow 0$; then while $x \,\&\, 1 = 0$, set $\rho \leftarrow \rho + 1$ and $x \leftarrow x \gg 1$." He reasoned that, when x is a random integer, the average number of right shifts is the average value of ρ, which is 1; and the standard deviation is only $\sqrt{2}$, so the loop almost always terminates quickly. Criticize his decision.

32. [*20*] What is the execution time for ρx when (52) is programmed for MMIX?

▶ **33.** [*26*] (Leiserson, Prokop, and Randall, 1998.) Show that if '58' is replaced by '49' in (52), we can use that method to identify *both* bits of the number $y = 2^j + 2^k$ quickly, when $64 > j > k \geq 0$. (Altogether $\binom{64}{2} = 2016$ cases need to be distinguished.)

34. [*M23*] Let x and y be 2-adic integers. Prove or disprove: $\rho x = \rho y$ if and only if $x \oplus y = (x-1) \oplus (y-1)$.

▶ **35.** [*M26*] According to Reitwiesner's theorem, exercise 4.1–34, every integer n has a unique representation $n = n^+ - n^-$ such that $\nu(n^+) + \nu(n^-)$ is minimized. Show that n^+ and n^- can be calculated quickly with bitwise operations. *Hint:* Prove the identity $(x \oplus 3x) \,\&\, ((x \oplus 3x) \gg 1) = 0$.

36. [*20*] Given $x = (x_{63} \ldots x_1 x_0)_2$, suggest efficient ways to calculate the quantities
- i) $x^{\oplus} = (x^{\oplus}_{63} \ldots x^{\oplus}_1 x^{\oplus}_0)_2$, where $x^{\oplus}_k = x_k \oplus \cdots \oplus x_1 \oplus x_0$ for $0 \leq k < 64$;
- ii) $x^{\&} = (x^{\&}_{63} \ldots x^{\&}_1 x^{\&}_0)_2$, where $x^{\&}_k = x_k \wedge \cdots \wedge x_1 \wedge x_0$ for $0 \leq k < 64$.

37. [*16*] What changes to (55) and (56) will make $\lambda 0$ come out -1?

38. [*17*] How long does the leftmost-bit-extraction procedure (57) take when implemented on MMIX?

▶ **39.** [*20*] Formula (43) shows how to remove the rightmost run of 1 bits from a given number x. How would you remove the *leftmost* run of 1 bits?

▶ **40.** [*21*] Prove (58), and find a simple way to decide if $\lambda x < \lambda y$, given x and $y \geq 0$.

41. [*M22*] What are the generating functions of the integer sequences (a) ρn, (b) λn, and (c) νn?

42. [*M21*] If $n = 2^{e_1} + \cdots + 2^{e_r}$, with $e_1 > \cdots > e_r \geq 0$, express the sum $\sum_{k=0}^{n-1} \nu k$ in terms of the exponents $e_1, \ldots, e_r$.

▶ **43.** [*20*] How sparse should x be, to make (63) faster than (62) on MMIX?

▶ **44.** [*23*] (E. Freed, 1983.) What's a fast way to evaluate the *weighted* bit sum $\sum j x_j$?

45. [*20*] (T. Rokicki, 1999.) Explain how to test if $x^R < y^R$, without reversing x and y.

46. [*22*] Method (68) uses six operations to interchange two bits $x_i \leftrightarrow x_j$ of a register. Show that this interchange can actually be done with only *three* MMIX instructions.

47. [*10*] Can the general δ-swap (69) also be done with a method like (67)?

48. [*M21*] How many different δ-swaps are possible in an n-bit register? (When $n = 4$, a δ-swap can transform 1234 into 1234, 1243, 1324, 1432, 2134, 2143, 3214, 3421, 4231.)

▶ **49.** [*M30*] Let $s(n)$ denote the fewest δ-swaps that suffice to reverse an n-bit number.
- a) Prove that $s(n) \geq \lceil \log_3 n \rceil$ when n is odd, $s(n) \geq \lceil \log_3 3n/2 \rceil$ when n is even.
- b) Evaluate $s(n)$ when $n = 3^m$, $2 \cdot 3^m$, $(3^m + 1)/2$, and $(3^m - 1)/2$.
- c) What are $s(32)$ and $s(64)$? *Hint:* Show that $s(5n + 2) \leq s(n) + 2$.

50. [*M37*] Continuing exercise 49, prove that $s(n) = \log_3 n + O(\log \log n)$.

∞
Presume
ruler function
analysis of algorithms
Leiserson
Prokop
Randall
2-adic integers
XOR identities
Reitwiesner
notation $x^{\oplus}$
suffix parity
prefix problem, see suffix parity
leftmost
run of 1 bits
[lg x]
generating functions
ρn
λn
νn
binary recurrence
nu(k) summed
Freed
weighted
sum of bits, weighted
Rokicki
reversing
interchange two bits

51. [*23*] Let c be a constant, $0 \le c < 2^d$. Find all sequences of masks $(\theta_0, \theta_1, \ldots, \theta_{d-1}, \hat\theta_{d-2}, \ldots, \hat\theta_1, \hat\theta_0)$ such that the general permutation scheme (71) takes $x \mapsto x^\pi$, where the bit permutation π is defined by either (a) $j\pi = j \oplus c$; or (b) $j\pi = (j + c) \bmod 2^d$. [The masks should satisfy $\theta_k \subseteq \mu_{d,k}$ and $\hat\theta_k \subseteq \mu_{d,k}$, so that (71) corresponds to Fig. 12; see (48). Notice that reversal, $x^\pi = x^R$, is the special case $c = 2^d - 1$ of part (a), while part (b) corresponds to the cyclic right shift $x^\pi = (x \gg c) + (x \ll (2^d - c))$.]

52. [*22*] Find hexadecimal constants $(\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \hat\theta_4, \hat\theta_3, \hat\theta_2, \hat\theta_1, \hat\theta_0)$ that cause (71) to produce the following important 64-bit permutations, based on the binary representation $j = (j_5j_4j_3j_2j_1j_0)_2$: (a) $j\pi = (j_0j_5j_4j_3j_2j_1)_2$; (b) $j\pi = (j_2j_1j_0j_5j_4j_3)_2$; (c) $j\pi = (j_1j_0j_5j_4j_3j_2)_2$; (d) $j\pi = (j_0j_1j_2j_3j_4j_5)_2$. [Case (a) is called a "perfect shuffle" because it takes $(x_{63} \ldots x_{33}x_{32}x_{31} \ldots x_1x_0)_2$ into $(x_{63}x_{31} \ldots x_{33}x_1x_{32}x_0)_2$; case (b) transposes an 8×8 matrix of bits; case (c), similarly, transposes a 16×4 matrix; and case (d) arises in connection with "fast Fourier transforms," see exercise 4.6.4–14.]

▸ **53.** [*M25*] The permutations in exercise 52 are said to be "induced by a permutation of index digits," because we obtain $j\pi$ by permuting the binary digits of j. Suppose $j\pi = (j_{(d-1)\psi} \ldots j_{1\psi}j_{0\psi})_2$, where ψ is a permutation of $\{0, 1, \ldots, d-1\}$. Prove that if ψ has t cycles, the 2^d-bit permutation $x \mapsto x^\pi$ can be obtained with only $d - t$ swaps. In particular, show that this observation speeds up all four cases of exercise 52.

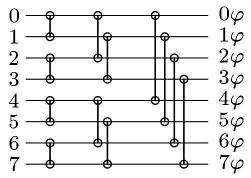
54. [*22*] (R. W. Gosper, 1985.) If an $m \times m$ bit matrix is stored in the rightmost m^2 bits of a register, show that it can be transposed by doing $(2^k(m-1))$-swaps for $0 \le k < \lceil \lg m \rceil$. Write out the method in detail when $m = 7$.

▸ **55.** [*26*] Suppose an $n \times n$ bit matrix is stored in the rightmost n^2 bits of an n^3-bit register. Prove that $18d + 2$ bitwise operations suffice to multiply two such matrices, when $n = 2^d$; the matrix multiplication can be either Boolean (like `MOR`) or mod 2 (like `MXOR`).

56. [*24*] Suggest a way to transpose a 7×9 bit matrix in a 64-bit register.

57. [*22*] Prove that any permutation of 2^d elements can be realized with the network $P(2^d)$ of Fig. 12 by some setting in which at most $d/(2d-1)$ of the crossbars are active.

▸ **58.** [*M27*] The first d columns of crossbar modules in the permutation network $P(2^d)$ perform a 1-swap, then a 2-swap, ..., and finally a 2^{d-1}-swap, when the wires of the network are stretched into horizontal lines as shown here for $d = 3$. Let $N = 2^d$. These N lines, together with the $Nd/2$ crossbars, form a so-called "Omega router." The purpose of this exercise is to study the set Ω of all permutations φ such that we can obtain $(0\varphi, 1\varphi, \ldots, (N-1)\varphi)$ as outputs on the right of an Omega router when the inputs at the left are $(0, 1, \ldots, N-1)$.



a) Prove that $|\Omega| = 2^{Nd/2}$. (Thus $\lg |\Omega| = Nd/2 \sim \frac{1}{2} \lg N!$.)

b) Prove that a permutation φ of $\{0, 1, \ldots, N-1\}$ belongs to Ω if and only if

$$i \bmod 2^k = j \bmod 2^k \;\text{ and }\; i\varphi \gg k = j\varphi \gg k \quad \text{implies} \quad i\varphi = j\varphi \qquad (*)$$

for all $0 \le i, j < N$ and all $0 \le k \le d$.

c) Simplify condition $(*)$ to the following, for all $0 \le i, j < N$:

$$\lambda(i\varphi \oplus j\varphi) < \rho(i \oplus j) \quad \text{implies} \quad i = j.$$

d) Let T be the set of all permutations τ of $\{0, 1, \ldots, N-1\}$ such that $\rho(i \oplus j) = \rho(i\tau \oplus j\tau)$ for all i and j. (This is the set of branching functions considered in exercise 14, modulo 2^d; so it has 2^{N-1} members, $2^{N/2+d-1}$ of which are the animating functions modulo 2^d.) Prove that $\varphi \in \Omega$ if and only if $\tau\varphi \in \Omega$ for all $\tau \in T$.

reversal
cyclic right shift
perfect shuffle
outshuffle
transposes
fast Fourier transforms
permutation of index digits
Gosper
transposed
matrix multiplication
Boolean matrix multiplication
`MOR`
`MXOR`
swap
Omega network for routing
butterfly network
shuffle network for routing
branching functions
animating functions

e) Suppose φ and ψ are permutations of Ω that operate on different elements; that is, $j\varphi \neq j$ implies $j\psi = j$, for $0 \leq j < N$. Prove that $\varphi\psi \in \Omega$.

59. [*M30*] Given $0 \leq a < b < N = 2^d$, how many Omega-routable permutations operate only on the interval $[a \,..\, b]$? (Thus we want to count the number of $\varphi \in \Omega$ such that $j\varphi \neq j$ implies $a \leq j \leq b$. Exercise 58(a) is the special case $a = 0$, $b = N - 1$.)

60. [*HM28*] Given a random permutation of $\{0, 1, \ldots, 2n{-}1\}$, let p_{nk} be the probability that there are 2^k ways to set the crossbars in the first and last columns of the permutation network $P(2n)$ when realizing this permutation. In other words, p_{nk} is the probability that the associated graph has k cycles (see (75)). What is the generating function $\sum_{k\geq 0} p_{nk} z^k$? What are the mean and variance of 2^k?

61. [*46*] Is it NP-hard to decide whether a given permutation is realizable with at least one mask $\theta_j = 0$, using the recursive method of Fig. 12 as implemented in (71)?

▸ **62.** [*22*] Let $N = 2^d$. We can obviously represent a permutation π of $\{0, 1, \ldots, N{-}1\}$ by storing a table of N numbers, d bits each. With this representation we have instant access to $y = x\pi$, given x; but it takes $\Omega(N)$ steps to find $x = y\pi^-$ when y is given.

Show that, with the same amount of memory, we can represent an arbitrary permutation in such a way that $x\pi$ and $y\pi^-$ are both computable in $O(d)$ steps.

63. [*16*] For what integers w, x, y, and z does the zipper function satisfy (i) $x \ddagger y = y \ddagger x$? (ii) $(x \ddagger y) \gg z = (x \gg \lceil z/2 \rceil) \ddagger (y \gg \lfloor z/2 \rfloor)$? (iii) $(w \ddagger x) \,\&\, (y \ddagger z) = (w \,\&\, y) \ddagger (x \,\&\, z)$?

64. [*22*] Find a "simple" expression for the zipper-of-sums $(x + x') \ddagger (y + y')$, as a function of $z = x \ddagger y$ and $z' = x' \ddagger y'$.

65. [*M16*] The binary polynomial $u(x) = u_0 + u_1 x + \cdots + u_{n-1}x^{n-1}$ (mod 2) can be represented by the integer $u = (u_{n-1} \ldots u_1 u_0)_2$. If $u(x)$ and $v(x)$ correspond to integers u and v in this way, what polynomial corresponds to $u \ddagger v$?

▸ **66.** [*M26*] Suppose the polynomial $u(x)$ has been represented as an n-bit integer u as in exercise 65, and let $v = u \oplus (u \ll \delta) \oplus (u \ll 2\delta) \oplus (u \ll 3\delta) \oplus \cdots$ for some integer δ.

a) What's a simple way to describe the polynomial $v(x)$?
b) Suppose n is large, and the bits of u have been packed into 64-bit words. How would you compute v when $\delta = 1$, using bitwise operations in 64-bit registers?
c) Consider the same question as (b), but when $\delta = 64$.
d) Consider the same question as (b), but when $\delta = 3$.
e) Consider the same question as (b), but when $\delta = 67$.

67. [*M31*] If $u(x)$ is a polynomial of degree $< n$, represented as in exercise 65, discuss the computation of $v(x) = u(x)^2 \bmod (x^n + x^m + 1)$, when $0 < m < n$ and both m and n are odd. *Hint:* This problem has an interesting connection with perfect shuffling.

68. [*20*] What three `MMIX` instructions implement the δ-shift operation, (79)?

69. [*25*] Prove that method (80) always extracts the proper bits when the masks θ_k have been set up properly: We never clobber any of the crucial bits y_j.

▸ **70.** [*31*] (Guy L. Steele Jr., 1994.) What's a good way to compute the masks θ_0, θ_1, $\ldots$, θ_{d-1} that are needed in the general compression procedure (80), given χ?

71. [*17*] Explain how to *reverse* the procedure of (80), going from the compact value $y = (y_{r-1} \ldots y_1 y_0)_2$ to a number $z = (z_{63} \ldots z_1 z_0)_2$ that has $z_{j_i} = y_i$ for $0 \leq i < r$.

72. [*10*] Simplify the expression $(x \ddagger y) \cdot\!|\!\cdot \mu_0$, when $x, y < 2^{2^{d-1}}$. (See Eqs. (76) and (81).)

73. [*22*] Prove that d sheep-and-goats steps will implement any 2^d-bit permutation.

permutation network
generating function
variance
NP-hard
represent an arbitrary permutation
zipper function
polynomial
polynomial remainder mod 2
trinomial
squaring a polynomial
perfect shuffling
MMIX
Steele
compression
unpacking
uncompressing
sheep-and-goats

74. [*22*] Given counts $(c_0, c_1, \ldots, c_{2^d-1})$ for the Chung–Wong procedure, explain why an appropriate cyclic 1-shift can always produce new counts $(c'_0, c'_1, \ldots, c'_{2^d-1})$ for which $\sum c'_{2l} = \sum c'_{2l+1}$, thus allowing the recursion to proceed.

▸ **75.** [*32*] The method of Chung and Wong replicates bit l of a register exactly c_l times, but it produces results in scrambled order. For example, the case $(c_0, \ldots, c_7) = (1, 2, 0, 2, 0, 2, 0, 1)$ illustrated in the text produces $(x_7x_0x_1x_5x_5x_3x_1x_3)_2$. In some applications this can be a disadvantage; we might prefer to have the bits retain their original order, namely $(x_7x_5x_5x_3x_3x_1x_1x_0)_2$ in that example.

Prove that the permutation network $P(2^d)$ of Fig. 12 can be modified to achieve this goal, given any sequence of counts $(c_0, c_1, \ldots, c_{2^d-1})$, if we replace the $d \cdot 2^{d-1}$ crossbar modules in the right-hand half by general 2×2 *mapping modules.* (A crossbar module with inputs (a, b) produces either (a, b) or (b, a) as output; a mapping module can also produce (a, a) or (b, b).)

76. [*47*] A *mapping network* is analogous to a sorting network or a permutation network, but it uses 2×2 mapping modules instead of comparators or crossbars, and it is supposed to be able to output all n^n possible mappings of its n inputs. Exercise 75, in conjunction with Fig. 12, shows that a mapping network for $n = 2^d$ exists with only $4d-2$ levels of delay, and with $n/2$ modules on each level; furthermore, this construction needs general 2×2 mapping modules (instead of simple crossbars) in only d of those levels.

To within $O(n)$, what is the smallest number $G(n)$ of modules that are sufficient to implement a general n-element mapping network?

77. [*26*] (R. W. Floyd and V. R. Pratt.) Design an algorithm that tests whether or not a given standard n-network is a sorting network, as defined in the exercises of Section 5.3.4. When the given network has r comparator modules, your algorithm should use $O(r)$ bitwise operations on words of length 2^n.

78. [*M27*] (*Testing disjointness.*) Suppose the binary numbers $x_1, x_2, \ldots, x_m$ each represent sets in a universe of $n - k$ elements, so that each x_j is less than 2^{n-k}. J. H. Quick (a student) decided to test whether the sets are disjoint by testing the condition

$$x_1 \mid x_2 \mid \cdots \mid x_m \;=\; (x_1 + x_2 + \cdots + x_m) \bmod 2^n.$$

Prove or disprove: Quick's test is valid if and only if $k \geq \lg(m-1)$.

▸ **79.** [*20*] If $x \neq 0$ and $x \subseteq \chi$, what is an easy way to determine the largest integer $x_\prime < x$ such that $x_\prime \subseteq \chi$? (Thus $(x_\prime)' = (x')_\prime = x$, in connection with (84).)

80. [*20*] Suggest a fast way to find all maximal proper subsets of a set. More precisely, given χ with $\nu\chi = m$, we want to find all $x \subseteq \chi$ such that $\nu x = m - 1$.

81. [*21*] Find a formula for "scattered difference," to go with the "scattered sum" (86).

82. [*21*] Is it easy to shift a scattered accumulator to the left by 1, for example to change $(y_2x_4x_3y_1x_2y_0x_1x_0)_2$ to $(y_1x_4x_3y_0x_2 0 x_1x_0)_2$?

▸ **83.** [*28*] Continuing exercise 82, find a way to shift a scattered 2^d-bit accumulator to the *right* by 1, given z and χ, in $O(d)$ steps.

84. [*25*] Given n-bit numbers $z = (z_{n-1} \ldots z_1z_0)_2$ and $\chi = (\chi_{n-1} \ldots \chi_1\chi_0)_2$, explain how to calculate the "stretched" quantities $z \leftharpoondown \chi = (z_{(n-1)\leftharpoondown\chi} \ldots z_{1\leftharpoondown\chi}z_{0\leftharpoondown\chi})_2$ and $z \rightharpoondown \chi = (z_{(n-1)\rightharpoondown\chi} \ldots z_{1\rightharpoondown\chi}z_{0\rightharpoondown\chi})_2$, where

$$j \leftharpoondown \chi = \max\{k \mid k \leq j \text{ and } \chi_k = 1\}, \qquad j \rightharpoondown \chi = \min\{k \mid k \geq j \text{ and } \chi_k = 1\};$$

Chung
Wong
replicates
mapping modules
crossbar module
mapping network
sorting network
distribution network, see mapping network
permutation network
Floyd
Pratt
sorting network
disjointness
represent sets
Quick
maximal proper subsets
scattered difference
scattered accumulator
scattered shifting
stretched
segmented broadcasting, see stretching

we let $z_{j\leftharpoondown\chi} = 0$ if $\chi_k = 0$ for $0 \le k \le j$, and $z_{j\rightharpoondown\chi} = 0$ if $\chi_k = 0$ for $n > k \ge j$. For example, if $n = 11$ and $\chi = (01101110010)_2$, then $z \leftharpoondown \chi = (z_9 z_9 z_8 z_6 z_6 z_5 z_4 z_1 z_1 z_1 0)_2$ and $z \rightharpoondown \chi = (0 z_9 z_8 z_8 z_6 z_5 z_4 z_4 z_4 z_1 z_1)_2$.

85. [*22*] (K. D. Tocher, 1954.) Imagine that you have a vintage 1950s computer with a drum memory for storing data, and that you need to do some computations with a $32 \times 32 \times 32$ array $a[i,j,k]$, whose subscripts are 5-bit integers in the range $0 \le i,j,k < 32$. Unfortunately your machine has only a very small high-speed memory: You can access only 128 consecutive elements of the array in fast memory at any time. Since your application usually moves from $a[i,j,k]$ to a neighboring position $a[i',j',k']$, where $|i-i'|+|j-j'|+|k-k'| = 1$, you have decided to allocate the array so that, if $i = (i_4 i_3 i_2 i_1 i_0)_2$, $j = (j_4 j_3 j_2 j_1 j_0)_2$, and $k = (k_4 k_3 k_2 k_1 k_0)_2$, the array entry $a[i,j,k]$ is stored in drum location $(k_4 j_4 i_4 k_3 j_3 i_3 k_2 j_2 i_2 k_1 j_1 i_1 k_0 j_0 i_0)_2$. By interleaving the bits in this way, a small change to i, j, or k will cause only a small change in the address.

Discuss implementation of this addressing function: (a) How does it change when i, j, or k changes by ± 1? (b) How would you handle a random access to $a[i,j,k]$, given i, j, and k? (c) How would you detect a "page fault" (namely, the condition that a new segment of 128 elements must be swapped into fast memory from the drum)?

86. [*M25*] An array of $2^p \times 2^q \times 2^r$ elements is to be allocated by putting $a[i,j,k]$ into a location whose bits are the $p+q+r$ bits of (i,j,k), permuted in some fashion. Furthermore, this array is to be stored in an external memory using pages of size 2^s. (Exercise 85 considers the case $p = q = r = 5$ and $s = 7$.) What allocation strategy of this kind minimizes the number of times that $a[i,j,k]$ is on a different page from $a[i',j',k']$, summed over all i, j, k, i', j', and k' such that $|i-i'|+|j-j'|+|k-k'| = 1$?

▶ **87.** [*20*] Suppose each byte of a 64-bit word x contains an ASCII code that represents either a letter, a digit, or a space. What three bitwise operations will convert all the lowercase letters to uppercase?

88. [*20*] Given $x = (x_7 \ldots x_0)_{256}$ and $y = (y_7 \ldots y_0)_{256}$, compute $z = (z_7 \ldots z_0)_{256}$, where $z_j = (x_j - y_j) \bmod 256$ for $0 \le j < 8$. (See the addition operation in (87).)

89. [*23*] Given $x = (x_{31} \ldots x_1 x_0)_4$ and $y = (y_{31} \ldots y_1 y_0)_4$, compute $z = (z_{31} \ldots z_1 z_0)_4$, where $z_j = \lfloor x_j / y_j \rfloor$ for $0 \le j < 32$, assuming that no y_j is zero.

90. [*20*] The bytewise averaging rule (88) always rounds downward when $x_j + y_j$ is odd. Make it less biased by rounding to the nearest odd integer in such cases.

▶ **91.** [*26*] (*Alpha channels.*) Recipe (88) is a good way to compute bytewise averages, but applications to computer graphics often require a more general blending of 8-bit values. Given three octabytes $x = (x_7 \ldots x_0)_{256}$, $y = (y_7 \ldots y_0)_{256}$, $\alpha = (a_7 \ldots a_0)_{256}$, show that bitwise operations allow us to compute $z = (z_7 \ldots z_0)_{256}$, where each byte z_j is a good approximation to $((255-a_j)x_j + a_j y_j)/255$, *without* doing any multiplication. Implement your method with `MMIX` instructions.

▶ **92.** [*21*] What happens if the second line of (88) is changed to '$z \leftarrow (x \mid y) - z$'?

93. [*18*] What basic formula for subtraction is analogous to formula (89) for addition?

94. [*21*] Let $x = (x_7 \ldots x_1 x_0)_{256}$ and $t = (t_7 \ldots t_1 t_0)_{256}$ in (90). Can t_j be nonzero when x_j is nonzero? Can t_j be zero when x_j is zero?

95. [*22*] What's a bitwise way to tell if all bytes of $x = (x_7 \ldots x_1 x_0)_{256}$ are distinct?

96. [*21*] Explain (93), and find a similar formula that sets test flags $t_j \leftarrow 128[x_j \le y_j]$.

97. [*23*] Leslie Lamport's paper in 1975 presented the following "problem taken from an actual compiler optimization algorithm": Given octabytes $x = (x_7 \ldots x_0)_{256}$ and $y =$

Tocher
allocate
storage allocation
interleaving the bits
page fault
ASCII code
lowercase letters
uppercase
multibyte subtraction
packed data+
division, 2-bit
averaging
rounding to the nearest odd
unbiased rounding
Alpha channels
subtraction
distinct
flags
Lamport

$(y_7 \ldots y_0)_{256}$, compute $t = (t_7 \ldots t_0)_{256}$ and $z = (z_7 \ldots z_0)_{256}$ so that $t_j \neq 0$ if and only if $x_j \neq 0$, $x_j \neq$ '*', and $x_j \neq y_j$; and $z_j = (x_j = 0?\ y_j\text{: } (x_j \neq \texttt{'*'} \wedge x_j \neq y_j?\ \texttt{'*'}\text{: } x_j))$.

98. [*20*] Given $x = (x_7 \ldots x_0)_{256}$ and $y = (y_7 \ldots y_0)_{256}$, compute $z = (z_7 \ldots z_0)_{256}$ and $w = (w_7 \ldots w_0)_{256}$, where $z_j = \max(x_j, y_j)$ and $w_j = \min(x_j, y_j)$ for $0 \le j < 8$.

▸ **99.** [*28*] Find hexadecimal constants a, b, c, d, e such that the six bitwise operations

$$y \leftarrow x \oplus a, \quad t \leftarrow ((((y \mathbin{\&} b) + c) \mid y) \oplus d) \mathbin{\&} e$$

will compute the flags $t = (f_7 \ldots f_1 f_0)_{256} \ll 7$ from any bytes $x = (x_7 \ldots x_1 x_0)_{256}$, where

$$f_0 = [x_0 = \texttt{'!'}],\ f_1 = [x_1 \neq \texttt{'*'}],\ f_2 = [x_2 < \texttt{'A'}],\ f_3 = [x_3 > \texttt{'z'}],\ f_4 = [x_4 \ge \texttt{'a'}],$$
$$f_5 = [x_5 \in \{\texttt{'0'}, \texttt{'1'}, \ldots, \texttt{'9'}\}],\ f_6 = [x_6 \le 168],\ f_7 = [x_7 \in \{\texttt{'<'}, \texttt{'='}, \texttt{'>'}, \texttt{'?'}\}].$$

100. [*25*] Suppose $x = (x_{15} \ldots x_1 x_0)_{16}$ and $y = (y_{15} \ldots y_1 y_0)_{16}$ are *binary-coded decimal* numbers, where $0 \le x_j, y_j < 10$ for each j. Explain how to compute their sum $u = (u_{15} \ldots u_1 u_0)_{16}$ and difference $v = (v_{15} \ldots v_1 v_0)_{16}$, where $0 \le u_j, v_j < 10$ and

$$(u_{15} \ldots u_1 u_0)_{10} = \bigl((x_{15} \ldots x_1 x_0)_{10} + (y_{15} \ldots y_1 y_0)_{10}\bigr) \bmod 10^{16},$$
$$(v_{15} \ldots v_1 v_0)_{10} = \bigl((x_{15} \ldots x_1 x_0)_{10} - (y_{15} \ldots y_1 y_0)_{10}\bigr) \bmod 10^{16},$$

without bothering to do any radix conversion.

▸ **101.** [*22*] Two octabytes x and y contain amounts of time, represented in five fields that respectively signify days (3 bytes), hours (1 byte), minutes (1 byte), seconds (1 byte), and milliseconds (2 bytes). Can you add and subtract them quickly, without converting from this mixed-radix representation to binary and back again?

102. [*25*] Discuss routines for the addition and subtraction of polynomials modulo 5, when (a) 16 4-bit coefficients or (b) 21 3-bit coefficients are packed into a 64-bit word.

▸ **103.** [*21*] Sometimes it's convenient to represent small numbers in *unary* notation, so that 0, 1, 2, 3, …, k appear respectively as $(0)_2$, $(1)_2$, $(11)_2$, $(111)_2$, …, $2^k - 1$ inside the computer. Then max and min are easily implemented as | and &.

Suppose the bytes of $x = (x_7 \ldots x_0)_{256}$ are such unary numbers, while the bytes of $y = (y_7 \ldots y_0)_{256}$ are all either 0 or 1. Explain how to "add" y to x or "subtract" y from x, giving $u = (u_7 \ldots u_0)_{256}$ and $v = (v_7 \ldots v_0)_{256}$ where

$$u_j = 2^{\min(8, \lg(x_j+1)+y_j)} - 1 \qquad \text{and} \qquad v_j = 2^{\max(0, \lg(x_j+1)-y_j)} - 1.$$

104. [*22*] Use bitwise operations to check the validity of a date represented in "year-month-day" fields (y, m, d) as in (22). You should compute a value t that is zero if and only if $1900 < y < 2100$, $1 \le m \le 12$, and $1 \le d \le \mathit{max_day}(m)$, where month m has at most $\mathit{max_day}(m)$ days. Can it be done in fewer than 20 operations?

105. [*30*] Given $x = (x_7 \ldots x_0)_{256}$ and $y = (y_7 \ldots y_0)_{256}$, discuss bitwise operations that will *sort* the bytes into order, so that $x_0 \le y_0 \le \cdots \le x_7 \le y_7$ afterwards.

106. [*27*] Explain the Fredman–Willard procedure (95). Also show that a simple modification of their method will compute $2^{\lambda x}$ without doing any left shifts.

▸ **107.** [*22*] Implement Algorithm B on MMIX when $d = 4$, and compare it with (56).

108. [*26*] Adapt Algorithm B to cases where n does not have the form $d \cdot 2^d$.

109. [*20*] Evaluate ρx for n-bit numbers x in $O(\log\log n)$ broadword steps.

multibyte max and min
comparison of bytes
bytes, testing relative order of
flags
binary-coded decimal
radix conversion
time
mixed-radix representation
polynomials modulo 5
unary
max
min
date
range checking
sort
Fredman
Willard
$2^{\lambda x}$+
binary log+
λx+
extract the most significant bit
MMIX
broadword
ruler function

▶ **110.** [*30*] Suppose $n = 2^{2^e}$ and $0 \le x < n$. Show how to compute $1 \ll x$ in $O(e)$ broadword steps, using only shift commands that shift by a constant amount. (Together with Algorithm B we can therefore extract the most significant bit of an n-bit number in $O(\log\log n)$ such steps.)

111. [*23*] Explain the 01^r pattern recognizer, (98).

112. [*46*] Can all occurrences of the pattern 1^r0 be identified in $O(1)$ broadword steps?

113. [*23*] A *strong broadword chain* is a broadword chain of a specified width n that is also a 2-adic chain, for all n-bit choices of x_0. For example, the 2-bit broadword chain (x_0, x_1) with $x_1 = x_0 + 1$ is not strong because $x_0 = (11)_2$ makes $x_1 = (00)_2$. But $(x_0, x_1, \ldots, x_4)$ is a strong broadword chain that computes $(x_0 + 1) \bmod 4$ for all $0 \le x_0 < 4$ if we set $x_1 = x_0 \oplus 1$, $x_2 = x_0 \mathbin{\&} 1$, $x_3 = x_2 \ll 1$, and $x_4 = x_1 \oplus x_3$.

Given a broadword chain $(x_0, x_1, \ldots, x_r)$ of width n, construct a strong broadword chain $(x'_0, x'_1, \ldots, x'_{r'})$ of the same width, such that $r' = O(r)$ and $(x_0, x_1, \ldots, x_r)$ is a subsequence of $(x'_0, x'_1, \ldots, x'_{r'})$.

114. [*16*] Suppose $(x_0, x_1, \ldots, x_r)$ is a strong broadword chain of width n that computes the value $f(x) = x_r$ whenever an n-bit number $x = x_0$ is given. Construct a broadword chain $(X_0, X_1, \ldots, X_r)$ of width mn that computes $X_r = (f(\xi_1)\ldots f(\xi_m))_{2^n}$ for any given mn-bit value $X_0 = (\xi_1 \ldots \xi_m)_{2^n}$, where $0 \le \xi_1, \ldots, \xi_m < 2^n$.

▶ **115.** [*24*] Given a 2-adic integer $x = (\ldots x_2x_1x_0)_2$, we might want to compute $y = (\ldots y_2y_1y_0)_2 = f(x)$ from x by zeroing out all blocks of consecutive 1s that (a) are not immediately followed by two 0s; or (b) are followed by an odd number of 0s before the next block of 1s begins; or (c) contain an odd number of 1s. For example, if x is $(\ldots 01110111001101000110)_2$ then y is (a) $(\ldots 00000111000001000110)_2$; (b) $(\ldots 00000111000000000110)_2$; (c) $(\ldots 00000000001100000110)_2$. (Infinitely many 0s are assumed to appear at the right of x_0. Thus, in case (a) we have

$$y_j = x_j \wedge \bigl((\bar{x}_{j-1}\wedge\bar{x}_{j-2}) \vee (x_{j-1}\wedge\bar{x}_{j-2}\wedge\bar{x}_{j-3}) \vee (x_{j-1}\wedge x_{j-2}\wedge\bar{x}_{j-3}\wedge\bar{x}_{j-4}) \vee \cdots\bigr)$$

for all j, where $x_k = 0$ for $k < 0$.) Find 2-adic chains for y in each case.

116. [*HM30*] Suppose $x = (\ldots x_2x_1x_0)_2$ and $y = (\ldots y_2y_1y_0)_2 = f(x)$, where y is computable by a 2-adic chain having no shift operations. Let L be the set of all binary strings such that $y_j = [x_j \ldots x_1x_0 \in L]$, and assume that all constants used in the chain are rational 2-adic numbers. Prove that L is a regular language. What languages L correspond to the functions in exercise 115(a) and 115(b)?

117. [*HM46*] Continuing exercise 116, is there any simple way to characterize the regular languages L that arise in shift-free 2-adic chains? (The language $L = 0^*(10^*10^*)^*$ does not seem to correspond to any such chain.)

118. [*30*] According to Lemma A, we cannot compute the function $x \gg 1$ for all n-bit numbers x by using only additions, subtractions, and bitwise Boolean operations (no shifts or branches). Show, however, that $O(n)$ such operations are necessary and sufficient if we include also the "monus" operator $y \mathbin{\dot{-}} z$ in our repertoire.

119. [*20*] Evaluate the function $f_{py}(x)$ in (102) with four broadword steps.

▶ **120.** [*M25*] There are $2^{n2^{mn}}$ functions that take n-bit numbers $(x_1, \ldots, x_m)$ into an n-bit number $f(x_1, \ldots, x_m)$. How many of them can be implemented with addition, subtraction, multiplication, and non-shift bitwise Boolean operations (modulo 2^n)?

▶ **121.** [*M25*] By exercise 3.1–6, a function from $[0\,.\,.\,2^n)$ into itself is eventually periodic.

extract the most significant bit pattern
strong broadword chain
broadword chain, strong
2-adic chains
rational 2-adic numbers
regular language
shifts
branches
monus

a) Prove that if f is any n-bit broadword function that can be implemented without shift instructions, the lengths of its periods are always powers of 2.
b) However, for every p between 1 and n, there's an n-bit broadword chain of length 3 that has a period of length p.

122. [*M22*] Complete the proof of Lemma B.

123. [*M23*] Let a_q be the constant $1 + 2^q + 2^{2q} + \cdots + 2^{(q-1)q} = (2^{q^2} - 1)/(2^q - 1)$. Using (104), show that there are infinitely many q such that the operation of multiplying by a_q, modulo 2^{q^2}, requires $\Omega(\log q)$ steps in any n-bit broadword chain with $n \ge q^2$.

124. [*M38*] Complete the proof of Theorem R$'$ by defining an n-bit broadword chain $(x_0, x_1, \ldots, x_f)$ and sets $(U_0, U_1, \ldots, U_f)$ such that, for $0 \le t \le f$, all inputs $x \in U_t$ lead to an essentially similar state $Q(x,t)$, in the following sense: (i) The current instruction in $Q(x,t)$ does not depend on x. (ii) If register r_j has a known value in $Q(x,t)$, it holds $x_{j'}$ for some definite index $j' \le t$. (iii) If memory location $M[z]$ has been changed, it holds $x_{z''}$ for some definite index $z'' \le t$. (The values of j' and z'' depend on j, z, and t, but not on x.) Furthermore $|U_t| \ge n/2^{2^t-1}$, and the program cannot guarantee that $r_1 = \rho x$ when $t < f$. *Hint:* Lemma B implies that a limited number of shift amounts and memory addresses need to be considered when t is small.

125. [*M33*] Prove Theorem P$'$. *Hint:* Lemma B remains true if we replace '$= 0$' by '$= \alpha_s$' in (103), for any values α_s.

126. [*M46*] Does the operation of extracting the most significant bit, $2^{\lambda x}$, require $\Omega(\log\log n)$ steps in an n-bit basic RAM? (See exercise 110.)

127. [*20*] Prove that if there's a way to carry out sideways addition of n-bit numbers in $O(\log\log n)$ broadword steps, then *every* symmetric function of a number's n bits can also be done in $O(\log\log n)$ broadword steps.

128. [*M46*] Does sideways addition require $\Omega(\log n)$ broadword steps?

129. [*M46*] Can the parity function $(\nu x) \bmod 2$ be computed in $O(1)$ broadword steps?

130. [*M46*] Is there an n-bit constant a such that the function $(a \ll x) \bmod 2^n$ requires $\Omega(\log n)$ n-bit broadword steps?

▸ **131.** [*23*] Write an `MMIX` program for Algorithm R when the graph is represented by arc lists. Vertex nodes have at least two fields, called `LINK` and `ARCS`, and arc nodes have `TIP` and `NEXT` fields, as explained in Section 7. Initially all `LINK` fields are zero, except in the given set of vertices Q, which is represented as a circular list. Your program should change that circular list so that it represents the set R of all reachable vertices.

▸ **132.** [*M27*] A *clique* in a graph is a set of mutually adjacent vertices; a clique is *maximal* if it's not contained in any other. The purpose of this exercise is to discuss an algorithm due to J. K. M. Moody and J. Hollis, which provides a convenient way to find every maximal clique of a not-too-large graph, using bitwise operations.

Suppose G is a graph with n vertices $V = \{0, 1, \ldots, n-1\}$. Let $\rho_v = \sum\{2^u \mid u \mathbin{—} v \text{ or } u = v\}$ be row v of G's reflexive adjacency matrix, and let $\delta_v = \sum\{2^u \mid u \ne v\} = 2^n - 1 - 2^v$. Every subset $U \subseteq V$ is representable as an n-bit integer $\sigma(U) = \sum_{u \in U} 2^u$; for example, $\delta_v = \sigma(V \setminus v)$. We also define the bitwise intersection

$$\tau(U) \;=\; \mathop{\&}_{0 \le u < n} (u \in U?\ \rho_u\colon \delta_u).$$

For example, if $n = 5$ we have $\tau(\{0,2\}) = \rho_0 \mathbin{\&} \delta_1 \mathbin{\&} \rho_2 \mathbin{\&} \delta_3 \mathbin{\&} \delta_4$.

period length
multiplying
broadword chain
extracting the most significant bit
basic RAM
sideways addition
symmetric function
parity function
circular list
graph algorithms+
cliques+
graph representation
set representation
maximal
Moody
Hollis
adjacency matrix

a) Prove that U is a clique if and only if $\tau(U) = \sigma(U)$.
b) Show that if $\tau(U) = \sigma(T)$ then T is a clique.
c) For $1 \le k \le n$, consider the 2^k bitwise intersections

$$C_k = \Big\{ \mathop{\&}_{0 \le u < k} (u \in U?\ \rho_u\colon \delta_u) \ \Big|\ U \subseteq \{0,1,\ldots,k-1\} \Big\},$$

and let C_k^+ be the maximal elements of C_k. Prove that U is a maximal clique if and only if $\sigma(U) \in C_n^+$.
d) Explain how to compute C_k^+ from C_{k-1}^+, starting with $C_0^+ = 2^n - 1$.

▸ **133.** [*20*] Given a graph G, how can the algorithm of exercise 132 be used to find (a) all maximal independent sets of vertices? (b) all minimal vertex covers (sets that hit every edge)?

134. [*15*] Nine classes of mappings for ternary values appear in (119), (123), and (124). To which class does the representation (128) belong, if $a = 0$, $b = *$, $c = 1$?

135. [*22*] Lukasiewicz included a few operations besides (127) in his three-valued logic: $\neg x$ (negation) interchanges 0 with 1 but leaves $*$ unchanged; $\diamond x$ (possibility) is defined as $\neg x \Rightarrow x$; $\Box x$ (necessity) is defined as $\neg\diamond\neg x$; and $x \Leftrightarrow y$ (equivalence) is defined as $(x \Rightarrow y) \wedge (y \Rightarrow x)$. Explain how to perform these operations using representation (128).

136. [*29*] Suggest two-bit encodings for binary operations on the set $\{a, b, c\}$ that are defined by the following "multiplication tables":

$$\text{(a)}\ \begin{pmatrix} a & b & c \\ b & c & c \\ c & c & c \end{pmatrix}; \qquad \text{(b)}\ \begin{pmatrix} a & c & b \\ c & b & a \\ b & a & c \end{pmatrix}; \qquad \text{(c)}\ \begin{pmatrix} a & b & a \\ a & a & c \\ a & b & c \end{pmatrix}.$$

137. [*21*] Show that the operation in exercise 136(c) is simpler with packed vectors like (131) than with the unpacked form (130).

138. [*24*] Find an example of three-state-to-two-bit encoding where class V_a is best.

139. [*25*] If x and y are signed bits 0, +1, or −1, what 2-bit encoding is good for calculating their sum $(z_1 z_2)_3 = x + y$, where z_1 and z_2 are also required to be signed bits? (This is a "half adder" for balanced ternary numbers.)

140. [*27*] Design an economical *full adder* for balanced ternary numbers: Show how to compute signed bits u and v such that $3u + v = x + y + z$ when $x, y, z \in \{0, +1, -1\}$.

▸ **141.** [*30*] The *Ulam numbers* $\langle U_1, U_2, \ldots\rangle = \langle 1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, \ldots\rangle$ are defined for $n \ge 3$ by letting U_n be the smallest integer $> U_{n-1}$ that has a *unique* representation $U_n = U_j + U_k$ for $0 < j < k < n$. Show that a million Ulam numbers can be computed rapidly with the help of bitwise techniques.

▸ **142.** [*33*] A subcube such as ∗10∗1∗01 can be represented by asterisk codes 10010100 and bit codes 01001001, as in (85); but many other encodings are also possible. What representation scheme for subcubes works best, for finding prime implicants by the consensus-based algorithm of exercise 7.1.1–31?

143. [*20*] Let x be a 64-bit number that represents an 8×8 chessboard, with a 1 bit in every position where a knight is present. Find a formula for the 64-bit number $f(x)$ that has a 1 in every position reachable in one move by a knight of x. For example, the white knights at the start of a game correspond to $x = \#42$; then $f(x) = \#\texttt{a51800}$.

144. [*16*] What node is the sibling of node j in a sideways heap? (See (134).)

145. [*17*] Interpret (137) when h is *less* than the height of j.

independent sets
vertex covers
mappings for ternary values
Lukasiewicz
three-valued logic
negation
possibility
necessity
equivalence
multiplication tables for groupoids
groupoids, mult tables for
packed
2-bit encoding
half adder
balanced ternary numbers
full adder
Ulam numbers
subcube
asterisk codes
bit codes
prime implicants
consensus
chessboard
knight
bit board
sibling
sideways heap+

▸ **146.** [*M20*] Prove Eq. (138), which relates the ρ and λ functions.

▸ **147.** [*M20*] What values of πv, βv, αv, and τj occur in Algorithm V when the forest is
a) the empty digraph with vertices $\{v_1, \ldots, v_n\}$ and no arcs?
b) the oriented path $v_n \to \cdots \to v_2 \to v_1$?

148. [*M21*] When preprocessing for Algorithm V, is it possible to have $\beta x_3 \to^* \beta y_2 \to^* \beta x_2 \to^* \beta y_1 \to^* \beta x_1$ in S when $x_3 \to x_2 \to x_1 \to \Lambda$ and $y_2 \to y_1 \to \Lambda$ in the forest? (If so, two different trees are "entangled" in S.)

▸ **149.** [*23*] Design a preprocessing procedure for Algorithm V.

▸ **150.** [*25*] Given an array of elements $A_1, \ldots, A_n$, the *range minimum query* problem is to determine $k(i,j)$ such that $A_{k(i,j)} = \min(A_i, \ldots, A_j)$ for any given indices i and j with $1 \le i \le j \le n$. Prove that Algorithm V will solve this problem, after $O(n)$ steps of preprocessing on the array A have prepared the necessary tables $(\pi, \beta, \alpha, \tau)$. *Hint:* Consider the binary search tree constructed from the sequence of keys $(p(1), p(2), \ldots, p(n))$, where p is a permutation of $\{1, 2, \ldots, n\}$ such that $A_{p(1)} \le A_{p(2)} \le \cdots \le A_{p(n)}$.

151. [*22*] Conversely, show that any algorithm for range minimum queries can be used to find nearest common ancestors, with essentially the same efficiency.

152. [*M21*] Prove that Algorithm V is correct.

▸ **153.** [*M20*] The pointers in a navigation pile like (144) can be packed into a binary string such as

0	1 0	0	1 0 0	0	0 0	1	0 1 0 0	0	0 0	0	0 0 0
2	4	6	8	10	12	14	16	18	20	22	24

.

At what bit position (from the left) does the pointer for node j end?

154. [*20*] The gray lines in Fig. 14 show how each pentagon is composed of ten triangles. What decomposition of the hyperbolic plane is defined by those gray lines alone, without the black pentagon edges?

▸ **155.** [*M21*] Prove that $(x\phi) \bmod 1 = (\alpha 0)_{1/\phi}$ when α is the negaFibonacci code for x.

156. [*21*] Design algorithms (a) to convert a given integer x to its negaFibonacci code α, and (b) to convert a given negaFibonacci code α to $x = N(\alpha)$.

157. [*M21*] Explain the recursion (148) for negaFibonacci predecessor and successor.

158. [*M26*] Let $\alpha = a_n \ldots a_1$ be the binary code for $F(\alpha 0) = a_n F_{n+1} + \cdots + a_1 F_2$ in the standard Fibonacci number system (146). Develop methods analogous to (148) and (149) for incrementing and decrementing such codewords.

159. [*M34*] Exercise 7 shows that it's easy to convert between the negabinary and binary number systems. Discuss conversion between negaFibonacci codewords and the ordinary Fibonacci codes in exercise 158.

160. [*M29*] Prove that (150) and (151) yield consistent code labels for the pentagrid.

161. [*20*] The cells of a chessboard can be colored black and white, so that neighboring cells have different colors. Does the pentagrid also have this property?

▸ **162.** [*HM37*] Explain how to draw the pentagrid, Fig. 14. What circles are present?

163. [*HM41*] Devise a way to navigate through the triangles in the tiling of Fig. 18.

164. [*23*] The original definition of custerization in 1957 was not (157) but

$$\text{custer}'(X) = X \mathbin{\&} {\sim}(X_{\rm NW} \mathbin{\&} X_{\rm N} \mathbin{\&} X_{\rm NE} \mathbin{\&} X_{\rm W} \mathbin{\&} X_{\rm E} \mathbin{\&} X_{\rm SW} \mathbin{\&} X_{\rm S} \mathbin{\&} X_{\rm SE}).$$

Why is (157) preferable?

ρ
λ
range minimum query
minimum element in subarray
binary search tree
nearest common ancestors
navigation pile
hyperbolic plane
tessellation
phi
negaFibonacci
Fibonacci number system
pentagrid
custerization+
bitmaps++++

165. [*21*] (R. A. Kirsch.) Discuss the computation of the 3×3 cellular automaton with

$$X^{(t+1)} = \text{custer}(\overline{X}^{(t)}) = {\sim}X^{(t)} \mathbin{\&} (X_{\rm N}^{(t)} \mid X_{\rm W}^{(t)} \mid X_{\rm E}^{(t)} \mid X_{\rm S}^{(t)}).$$

166. [*M23*] Let $f(M,N)$ be the maximum number of black pixels in an $M \times N$ bitmap X for which $X = \text{custer}(X)$. Prove that $f(M,N) = \frac{4}{5}MN + O(M+N)$.

167. [*24*] (*Life.*) If the bitmap X represents an array of cells that are either dead (0) or alive (1), the Boolean function

$$f(x_{\rm NW},\ldots,x,\ldots,x_{\rm SE}) = [2 < x_{\rm NW}+x_{\rm N}+x_{\rm NE}+x_{\rm W}+\tfrac{1}{2}x+x_{\rm E}+x_{\rm SW}+x_{\rm S}+x_{\rm SE} < 4]$$

can lead to astonishing life histories when it governs a cellular automaton as in (158).

a) Find a way to evaluate f with a Boolean chain of 26 steps or less.

b) Let $X_j^{(t)}$ denote row j of X at time t. Show that $X_j^{(t+1)}$ can be evaluated in at most 23 broadword steps, as a function of the three rows $X_{j-1}^{(t)}$, $X_j^{(t)}$, and $X_{j+1}^{(t)}$.

▶ **168.** [*23*] To keep an image finite, we might insist that a 3 × 3 cellular automaton treats a $M \times N$ bitmap as a *torus*, wrapping around seamlessly between top and bottom and between left and right. The task of simulating its actions efficiently with bitwise operations is somewhat tricky: We want to minimize references to memory, yet each new pixel value depends on old values that lie on all sides. Furthermore the shifting of bits between neighboring words tends to be awkward, taxing the capacity of a register.

Show that such difficulties can be surmounted by maintaining an array of n-bit words A_{jk} for $0 \le j \le M$ and $0 \le k \le N' = \lceil N/(n-2)\rceil$. If $j \ne M$ and $k \ne 0$, word A_{jk} should contain the pixels of row j and columns $(k-1)(n-2)$ through $k(n-2)+1$, inclusive; the other words A_{Mk} and A_{j0} provide auxiliary buffer space. (Notice that some bits of the raster appear twice.)

169. [*22*] Continuing the previous two exercises, what happens to the Cheshire cat of Fig. 17(a) when it is subjected to the vicissitudes of Life, in a 26 × 31 torus?

▶ **170.** [*21*] What result does the Guo–Hall thinning automaton produce when given a solid black rectangle of M rows and N columns? How long does it take?

171. [*24*] Find a Boolean chain of length ≤ 25 to evaluate the local thinning function $g(x_{\rm NW}, x_{\rm N}, x_{\rm NE}, x_{\rm W}, x_{\rm E}, x_{\rm SW}, x_{\rm S}, x_{\rm SE})$ of (159), with or without the extra cases in (160).

172. [*M29*] Prove or disprove: If a pattern contains three black pixels that are king-neighbors of each other, the Guo–Hall procedure extended by (160) will reduce it, unless none of those pixels can be removed without destroying the connectivity.

▶ **173.** [*M30*] Raster images often need to be cleaned up if they contain noisy data. For example, accidental specks of black or white may well spoil the results when a thinning algorithm is used for optical character recognition.

Say that a bitmap X is *closed* if every white pixel is part of a 2 × 2 square of white pixels, and *open* if every black pixel is part of a 2 × 2 square of black pixels. Let

$$X^D = \mathop{\&}\{Y \mid Y \supseteq X \text{ and } Y \text{ is closed}\}; \quad X^L = \big|\{Y \mid Y \subseteq X \text{ and } Y \text{ is open}\}.$$

A bitmap is called *clean* if it equals X^{DL} for some X. We might, for example, have

$X =$  ; $X^D =$  ; $X^{DL} =$  .

In general X^D is "darker" than X, while X^L is "lighter": $X^D \supseteq X \supseteq X^L$.

a) Prove that $(X^{DL})^{DL} = X^{DL}$. *Hint:* $X \subseteq Y$ implies $X^D \subseteq Y^D$ and $X^L \subseteq Y^L$.

b) Show that X^D can be computed with one step of a 3 × 3 cellular automaton.

Kirsch
cellular automaton
Life
broadword
torus
Cheshire cat
Life
Guo
Hall
thinning
noisy data
thinning
optical character recognition
closed
open
clean

174. [*M46*] (M. Minsky and S. Papert.) Is there a three-dimensional shrinking algorithm that preserves connectivity, analogous to (161)?

175. [*15*] How many *rookwise* connected black components does the Cheshire cat have?

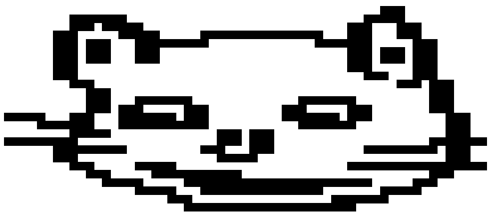
176. [*M24*] Let G be the graph whose vertices are the black pixels of a given bitmap X, with $u \mathrel{-\!\!-} v$ when u and v are a king move apart. Let G' be the corresponding graph after the shrinking transformation (161) has been applied. The purpose of this exercise is to show that the number of connected components of G' is the number of components of G minus the number of isolated vertices of G.

Let $N_{(i,j)} = \{(i,j),(i-1,j),(i-1,j+1),(i,j+1)\}$ be pixel (i,j) together with its north and/or east neighbors. For each $v \in G$ let $S(v) = \{v' \in G' \mid v' \in N_v\}$.

a) Prove that $S(v)$ is empty if and only if v is isolated in G.
b) If $u \mathrel{-\!\!-} v$ in G, $u' \in S(u)$, and $v' \in S(v)$, prove that $u' \mathrel{-\!\!-}^* v'$ in G'.
c) For each $v' \in G'$ let $S'(v') = \{v \in G \mid v' \in N_v\}$. Is $S'(v')$ always nonempty?
d) If $u' \mathrel{-\!\!-} v'$ in G', $u \in S'(u')$, and $v \in S'(v')$, prove that $u \mathrel{-\!\!-}^* v$ in G.
e) Hence there's a one-to-one correspondence between the nontrivial components of G and the components of G'.

177. [*M22*] Continuing exercise 176, prove an analogous result for the white pixels.

178. [*20*] If X is an $M \times N$ bitmap, let X^* be the $M \times (2N+1)$ bitmap $X \ddagger (X \mid (X \ll 1))$. Show that the kingwise connected components of X^* are also rookwise connected, and that bitmap X^* has the same "surroundedness tree" (162) as X.



▸ **179.** [*34*] Design an algorithm that constructs the surroundedness tree of a given $M \times N$ bitmap, scanning the image one row at a time as discussed in the text. (See (162) and (163).)

▸ **180.** [*M24*] Digitize the hyperbola $y^2 = x^2 + 13$ by hand, for $0 < y \le 7$.

181. [*HM20*] Explain how to subdivide a general conic (168) with rational coefficients into monotonic parts so that Algorithm T applies.

182. [*M31*] Why does the three-register method (Algorithm T) digitize correctly?

▸ **183.** [*M22*] Find a quadratic form $Q'(x,y)$ so that, when Algorithm T is applied to (x',y'), (x,y), and Q', it produces exactly the same edges as it does from (x,y), (x',y'), and Q, but in the reverse order.

▸ **184.** [*22*] Design an algorithm that properly digitizes a straight line from (ξ,η) to (ξ',η'), when ξ, η, ξ', and η' are rational numbers, by simplifying Algorithm T.

185. [*HM22*] Given three complex numbers (z_0,z_1,z_2), consider the curve traced out by

$$B(t) \;=\; (1-t)^2 z_0 + 2(1-t)t z_1 + t^2 z_2, \qquad \text{for } 0 \le t \le 1.$$

a) What is the approximate behavior of $B(t)$ when t is near 0 or 1?
b) Let $S(z_0,z_1,z_2) = \{B(t) \mid 0 \le t \le 1\}$. Prove that all points of $S(z_0,z_1,z_2)$ lie on or inside the triangle whose vertices are z_0, z_1, and z_2.
c) True or false? $S(w+\zeta z_0, w+\zeta z_1, w+\zeta z_2) = w + \zeta S(z_0,z_1,z_2)$.
d) Prove that $S(z_0,z_1,z_2)$ is part of a straight line if and only if z_0, z_1, and z_2 are collinear; otherwise it is part of a parabola.

Minsky
Papert
shrinking
Cheshire cat
zipper function
kingwise connected
rookwise connected
surroundedness tree
hyperbola
conic
monotonic parts
three-register method
straight line
Bézier splines+
squines
parabola

e) Prove that if $0 \le \theta \le 1$, we have the recurrence

$$S(z_0, z_1, z_2) \;=\; S(z_0, (1{-}\theta)z_0 + \theta z_1, B(\theta)) \cup S(B(\theta), (1{-}\theta)z_1 + \theta z_2, z_2).$$

186. [*M29*] Continuing exercise 185, show how to digitize $S(z_0, z_1, z_2)$ using the three-register method (Algorithm T). For best results, the digitizations of $S(z_2, z_1, z_0)$ and $S(z_0, z_1, z_2)$ should produce the same edges, but in reverse order.

▶ **187.** [*25*] Bitmap images can often be viewed conveniently using pixels that are *shades of gray* instead of just black or white. Such gray levels typically are 8-bit values that range from 0 (black) to 255 (white); notice that the black/white convention is traditionally *reversed* with respect to the 1-bit case. An $m \times n$ bitmap whose resolution is 600 dots per inch corresponds nicely to the $(m/8) \times (n/8)$ grayscale image with 75 pixels per inch that is obtained by mapping each 8×8 subarray of 1-bit pixels into the gray level $\lfloor 255(1 - k/64)^{1/\gamma} + \frac{1}{2} \rfloor$, where $\gamma = 1.3$ and k is the number of 1s in the subarray.

Write an `MMIX` routine that converts a given $m \times n$ array `BITMAP` into the corresponding $(m/8) \times (n/8)$ image `GRAYMAP`, assuming that m is a multiple of 8 and that n is a multiple of 64.

188. [*25*] Given a 64×64 bitmap, what's a good way (a) to transpose it, or (b) to rotate it by 90°, using operations on 64-bit numbers?

189. [*23*] A *parity pattern* of length m and width n is an $m \times n$ matrix of 0s and 1s with the property that each element is the sum of its neighbors, mod 2. For example,

$$\begin{array}{c}11\\00\\11\end{array},\quad \begin{array}{c}0011\\0100\\1101\\0101\end{array},\quad \begin{array}{c}01010\\11011\\01010\end{array},\quad \begin{array}{c}100\\110\\101\\011\\001\end{array},\quad \text{and}\quad \begin{array}{c}01110\\10101\\11011\\10101\\01110\end{array}$$

are parity patterns of sizes 3×2, 4×4, 3×5, 5×3, and 5×5.

a) If the binary vectors $\alpha_1, \alpha_2, \ldots, \alpha_m$ are the rows of a parity pattern, show that $\alpha_2, \ldots, \alpha_m$ can all be computed from the top row α_1 by using bitwise operations. Thus at most one $m \times n$ parity pattern can begin with any given bit vector.

b) True or false: The sum (mod 2) of two $m \times n$ parity patterns is a parity pattern.

c) A parity pattern is called *perfect* if it contains no all-zero row or column. For example, three of the matrices above are perfect, but the 3×2 and 3×5 examples are not. Show that every $m \times n$ parity pattern contains a perfect parity pattern as a submatrix. Furthermore, all such submatrices have the same size, $m' \times n'$, where $m' + 1$ is a divisor of $m + 1$ and $n' + 1$ is a divisor $n + 1$.

d) There's a perfect parity pattern whose first row is 0011, but there is no such pattern beginning with 01010. Is there a simple way to decide whether a given binary vector is the top row of a perfect parity pattern?

e) Prove that there's a unique perfect parity pattern that begins with $1\overbrace{0\ldots0}^{n-1}$.

190. [*M30*] A *wraparound parity pattern* is analogous to the parity patterns of exercise 189, except that the leftmost and rightmost elements of each row are also neighbors.

a) Find a simple relation between the parity pattern of width n that begins with α and the wraparound parity pattern of width $2n + 2$ that begins with $0\alpha 0\alpha^R$.

b) The Fibonacci polynomials $F_j(x)$ are defined by the recurrence

$$F_0(x) = 0, \qquad F_1(x) = 1, \qquad \text{and} \qquad F_{j+1}(x) = xF_j(x) + F_{j-1}(x) \quad \text{for } j \ge 1.$$

pixels
shades of gray
gray levels
black
white
transpose
rotate
parity pattern+
perfect parity pattern+
0–1 matrix, see also bitmap
Boolean matrix, see also bitmap
wraparound parity pattern
recurrence
Fibonacci polynomials+

polynomial remainder mod 2
fractals
0–1 matrix
rank
triangularization
null spaces
Thompson
multibyte encoding+
lexicographic order

Show that there's a simple relation between the wraparound parity patterns that begin with $10\ldots0$ ($N-1$ zeros) and the Fibonacci polynomials modulo x^N+1. *Hint:* Consider $F_j(x^{-1}+1+x)$, and do arithmetic mod 2 as well as mod x^N+1.

c) If α is the binary string $a_1\ldots a_n$, let $f_\alpha(x)=a_1x+\cdots+a_nx^n$. Show that

$$f_{(\alpha_j 0\, \alpha_j^R)}(x) \;=\; (f_\alpha(x)+f_\alpha(x^{-1}))F_j(x^{-1}{+}1{+}x) \bmod (x^N+1) \text{ and mod } 2,$$

when $N=2n+2$ and α_j is row j of a width-n parity pattern that begins with α.

d) Consequently we can compute α_j from α in only $O(n^2\log j)$ steps. *Hints:* See exercise 4.6.3–26; and use the identity $F_{m+n}(x)=F_m(x)F_{n+1}(x)+F_{m-1}(x)F_n(x)$, which generalizes Eq. 1.2.8–(6).

191. [*HM38*] The shortest parity pattern that begins with a given string can be quite long; for example, it turns out that the perfect pattern of width 120 whose first row is $10\ldots0$ has length 36,028,797,018,963,966(!). The purpose of this exercise is to consider how to calculate the interesting function

$$c(q)=1+\max\{\,m\mid \text{there exists a perfect parity pattern of length } m \text{ and width } q{-}1\},$$

whose initial values $(1,3,4,6,5,24,9,12,28)$ for $1\le q\le 9$ are easy to compute by hand.

a) Characterize $c(q)$ algebraically, using the Fibonacci polynomials of exercise 190.

b) Explain how to calculate $c(q)$ if we know a number M such that $c(q)$ divides M, and if we also know the prime factors of M.

c) Prove that $c(2^e)=3\cdot2^{e-1}$ when $e>0$. *Hint:* $F_{2^e}(y)$ has a simple form, mod 2.

d) Prove that when q is odd and not a multiple of 3, $c(q)$ is a divisor of $2^{2e}-1$, where e is the order of 2 modulo q. *Hint:* $F_{2^e-1}(y)$ has a simple form, mod 2.

e) What happens when q is an odd multiple of 3?

f) Finally, explain how to handle the case when q is even.

▸ **192.** [*M21*] If a perfect $m\times n$ parity pattern exists, when m and n are odd, show that there's also a perfect $(2m{+}1)\times(2n{+}1)$ parity pattern. (Intricate fractals arise when this observation is applied repeatedly; for example, the 5×5 pattern in exercise 189 leads to Fig. 20.)

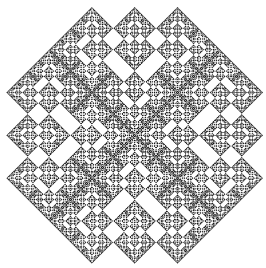


Fig. 20. A perfect 383×383 parity pattern.

193. [*24*] Find all $n\le383$ for which there exists a perfect $n\times n$ parity pattern with 8-fold symmetry, such as the example in Fig. 20. *Hint:* The diagonal elements of all such patterns must be zero.

▸ **194.** [*HM25*] Let A be a binary matrix having rows $\alpha_1,\ldots,\alpha_m$ of length n. Explain how to use bitwise operations to compute the rank $m-r$ of A over the binary field $\{0,1\}$, and to find linearly independent binary vectors $\theta_1,\ldots,\theta_r$ of length m such that $\theta_jA=0\ldots0$ for $1\le j\le r$. *Hint:* See the "triangularization" algorithm for null spaces, Algorithm 4.6.2N.

195. [*21*] (K. Thompson, 1992.) Integers in the range $0\le x<2^{31}$ can be encoded as a string of up to six bytes $\alpha(x)=\alpha_1\ldots\alpha_l$ in the following way: If $x<2^7$, set $l\leftarrow1$ and $\alpha_1\leftarrow x$. Otherwise let $x=(x_5\ldots x_1x_0)_{64}$; set $l\leftarrow\lceil(\lambda x)/5\rceil$, $\alpha_1\leftarrow2^8-2^{8-l}+x_{l-1}$, and $\alpha_j=2^7+x_{l-j}$ for $2\le j\le l$. Notice that $\alpha(x)$ contains a zero byte if and only if $x=0$.

a) What are the encodings of #`a`, #`3a3`, #`7b97`, and #`1D141`?

b) If $x\le x'$, prove that $\alpha(x)\le\alpha(x')$ in lexicographic order.

c) Suppose a sequence of values $x^{(1)}x^{(2)}\ldots x^{(n)}$ has been encoded as a byte string $\alpha(x^{(1)})\alpha(x^{(2)})\ldots\alpha(x^{(n)})$, and let α_k be the kth byte in that string. Show that

it's easy to determine the value $x^{(i)}$ from which α_k came, by looking at a few of the neighboring bytes if necessary.

196. [*22*] The Universal Character Set (UCS), also known as Unicode, is a standard mapping of characters to integer codepoints x in the range $0 \le x < 2^{20} + 2^{16}$. An encoding called UTF-16 represents such integers as one or two wydes $\beta(x) = \beta_1$ or $\beta(x) = \beta_1\beta_2$, in the following way: If $x < 2^{16}$ then $\beta(x) = x$; otherwise

$$\beta_1 = {}^{\#}\mathtt{d800} + \lfloor y/2^{10} \rfloor \text{ and } \beta_2 = {}^{\#}\mathtt{dc00} + (y \bmod 2^{10}), \text{ where } y = x - 2^{16}.$$

Answer questions (a), (b), and (c) of exercise 195 for this encoding.

▸ **197.** [*21*] Unicode characters are often represented as strings of bytes using a scheme called UTF-8, which is the encoding of exercise 195 restricted to integers in the range $0 \le x < 2^{20}+2^{16}$. Notice that UTF-8 efficiently preserves the standard ASCII character set (the codepoints with $x < 2^7$), and that it is quite different from UTF-16.

Let α_1 be the first byte of a UTF-8 string $\alpha(x)$. Show that there are reasonably small integer constants a, b, and c such that only four bitwise operations

$$(a \gg ((\alpha_1 \gg b) \mathbin{\&} c)) \mathbin{\&} 3$$

suffice to determine the number $l - 1$ of bytes between α_1 and the end of $\alpha(x)$.

▸ **198.** [*23*] A person might try to encode #`a` as #`c08a` or #`e0808a` or #`f080808a` in UTF-8, because the obvious decoding algorithm produces the same result in each case. But such unnecessarily long forms are illegal, because they could lead to security holes.

Suppose α_1 and α_2 are bytes such that $\alpha_1 \ge {}^{\#}\mathtt{80}$ and ${}^{\#}\mathtt{80} \le \alpha_2 < {}^{\#}\mathtt{c0}$. Find a branchless way to decide whether α_1 and α_2 are the first two bytes of at least one legitimate UTF-8 string $\alpha(x)$.

199. [*20*] Interpret the contents of register \$3 after the following three `MMIX` instructions have been executed: `MOR $1,$0,#94; MXOR $2,$0,#94; SUBU $3,$2,$1`.

200. [*20*] Suppose $x = (x_{15} \ldots x_1 x_0)_{16}$ has sixteen hexadecimal digits. What one `MMIX` instruction will change each nonzero digit to `f`, while leaving zeros untouched?

201. [*20*] What two instructions will change an octabyte's nonzero wydes to #`ffff`?

202. [*22*] Suppose we want to convert a tetrabyte $x = (x_7 \ldots x_1 x_0)_{16}$ to the octabyte $y = (y_7 \ldots y_1 y_0)_{256}$, where y_j is the ASCII code for the hexadecimal digit x_j. For example, if $x = {}^{\#}\mathtt{1234abcd}$, y should represent the 8-character string `"1234abcd"`. What clever choices of five constants `a`, `b`, `c`, `d`, and `e` will make the following `MMIX` instructions do the job?

```
MOR t,x,a;  SLU s,t,4;  XOR t,s,t;  AND t,t,b;
ADD t,t,c;  MOR s,d,t;  ADD t,t,e;  ADD y,t,s.
```

▸ **203.** [*22*] What are the amazing constants p, q, r, m that achieve a perfect shuffle with just six `MMIX` commands? (See (175)–(178).)

204. [*20*] The perfect shuffle (175) is sometimes called an "outshuffle," by comparison with the "inshuffle" that takes $z \mapsto y \ddagger x = (y_{31}x_{31} \ldots y_1 x_1 y_0 x_0)_2$; the outshuffle preserves the leftmost and rightmost bits of z, but the inshuffle has no fixed points. Can an inshuffle be performed as efficiently as an outshuffle?

▸ **205.** [*23*] What's a fast way for `MMIX` to transpose an 8×8 Boolean matrix?

▸ **206.** [*21*] Is the suffix parity operation $x^{\oplus}$ of exercise 36 easy to compute with `MXOR`?

Universal Character Set
UCS
Unicode
UTF-16: 16-bit UCS Transformation Forma
UTF-8
ASCII
packing
fractional precision
table lookup by shifting
wyde: a 16-bit quantity
byte: an 8-bit quantity
nybble: a 4-bit quantity
nyp: a 2-bit quantity
tetrabyte or tetra: a 32-bit quantity
octabyte or octa: a 64-bit quantity
security
branchless
MOR+
MXOR+
hexadecimal digits
masks
ASCII
hexadecimal digit
perfect shuffle
outshuffle
inshuffle
transpose
Boolean matrix
suffix parity

207. [*22*] A puzzle: Register `x` contains a number $8j+k$, where $0 \le j, k < 8$. Registers `a` and `b` contain arbitrary octabytes $(a_7 \ldots a_1 a_0)_{256}$ and $(b_7 \ldots b_1 b_0)_{256}$. Find a sequence of four `MMIX` instructions that will put $a_j \mathbin{\&} b_k$ into register `x`.

▸ **208.** [*M25*] The truth table of a Boolean function $f(x_1, \ldots, x_6)$ is essentially a 64-bit number

$$f = \bigl(f(0,0,0,0,0,0) \ldots f(1,1,1,1,1,0) f(1,1,1,1,1,1)\bigr)_2.$$

Show that two `MOR` instructions will convert f to the truth table of $\hat{f}$, the least monotone Boolean function that is greater than or equal to f at each point.

209. [*M32*] Suppose $a = (a_{63} \ldots a_1 a_0)_2$ represents the polynomial

$$a(x) = (a_{63} \ldots a_1 a_0)_x = a_{63}x^{63} + \cdots + a_1 x + a_0.$$

Discuss using `MXOR` to compute the product $c(x) = a(x)b(x)$, modulo x^{64} and mod 2.

▸ **210.** [*HM26*] Implement the CRC procedure (183) on `MMIX`.

▸ **211.** [*HM28*] (R. W. Gosper.) Find a short, branchless `MMIX` computation that will compute the inverse of any given 8×8 matrix X of 0s and 1s, modulo 2, assuming that $\det X$ is odd.

truth table
monotone Boolean function
polynomial multiplication
MXOR
CRC
Gosper
branchless
inverse
matrix X of 0s and 1s

SECTION 7.1.3

interchange
swap
Warren
unsigned 2-adic integers
magic
HAKMEM
Agrawal
two's complement
binary basis
Sprague
Grundy
Bouton
commutative laws

1. These operations interchange the bits of x and y in positions where m is 1. (In particular, if $m = -1$, the step '$y \leftarrow y \oplus (x \,\&\, m)$' becomes just '$y \leftarrow y \oplus x$', and the three assignments will swap $x \leftrightarrow y$ without needing an auxiliary register. H. S. Warren, Jr., has located this trick in vintage-1961 IBM programming course notes.)

2. All three hold when x and y are nonnegative, or if we regard x and y as "unsigned 2-adic integers" in which $0 < 1 < 2 < \cdots < -3 < -2 < -1$. But if negative integers are less than nonnegative integers, (i) fails if and only if $x < 0$ and $y < 0$; (ii) and (iii) fail if and only if $x \oplus y < 0$, namely, if and only if $x < 0$ and $y \ge 0$ or $x \ge 0$ and $y < 0$.

3. Note that $x - y = (x \oplus y) - 2(\bar{x} \,\&\, y)$ (see exercise 93). By removing bits common to x and y at the left, we may assume that $x_{n-1} = 1$ and $y_{n-1} = 0$. Then $2(\bar{x} \,\&\, y) \le 2((x \oplus y) - 2^{n-1}) = (x \oplus y) - (x \oplus y)^M - 1$.

4. $x^{CN} = x + 1 = x^S$, by (16). Hence $x^{NC} = x^{NCSP} = x^{NCCNP} = x^{NNP} = x^P$.

5. (a) Disproof: Let $x = (\dots x_2 x_1 x_0)_2$. Then digit l of $x \ll k$ is $x_{l-k}[l \ge k]$. So digit l of the left-hand side is $x_{l-k-j}[l \ge k][l - k \ge j]$, while digit l of the right-hand side is $x_{l-j-k}[l \ge j + k]$. These expressions agree if $j \ge 0$ or $k \le 0$. But if $j < 0 < k$, they differ when $l = \max(0, j + k)$ and $x_{l-j-k} = 1$.

(We do, however, have $(x \ll j) \ll k \subseteq x \ll (j + k)$ in all cases.)

(b) Proof: Digit l in all three formulas is $x_{l+j}[l \ge -j] \wedge y_{l-k}[l \ge k]$.

6. Since $x \ll y \ge 0$ if and only if $x \ge 0$, we must have $x \ge 0$ if and only if $y \ge 0$. Obviously $x = y$ is always a solution. The solutions with $x > y$ are (a) $x = -1$ and $y = -2$, or $2^y > x > y > 0$; (b) $x = 2$ and $y = 1$, or $2^{-x} \ge -y > -x > 0$.

7. Set $x' \leftarrow (x + \bar{\mu}_0) \oplus \bar{\mu}_0$, where μ_0 is the constant in (47). Then $x' = (\dots x'_2 x'_1 x'_0)_2$, since $(x' \oplus \bar{\mu}_0) - \bar{\mu}_0 = (\dots \bar{x}'_3 x'_2 \bar{x}'_1 x'_0)_2 - (\dots 1010)_2 = (\dots 0x'_2 0x'_0)_2 - (\dots x'_3 0 x'_1 0)_2 = x$.

[This is Hack 128 in HAKMEM; see answer 20 below. An alternative formula, $x' \leftarrow (\mu_0 - x) \oplus \mu_0$, has also been suggested by D. P. Agrawal, *IEEE Trans.* **C-29** (1980), 1032–1035. The results are correct modulo 2^n for all n, but overflow or underflow can occur. For example, two's complement binary numbers in an n-bit register range from -2^{n-1} to $2^{n-1} - 1$, inclusive, but negabinary numbers range from $-\frac{2}{3}(2^n - 1)$ to $\frac{1}{3}(2^n - 1)$ when n is even. In general the formula $x' \leftarrow (x + \mu) \oplus \mu$ converts from binary notation to the general number system with binary basis $\langle 2^n(-1)^{m_n} \rangle$ discussed in exercise 4.1–30(c), when $\mu = (\dots m_2 m_1 m_0)_2$.]

8. First, $x \oplus y \notin (S \oplus y) \cup (x \oplus T)$. Second, suppose that $0 \le k < x \oplus y$, and let $x \oplus y = (\alpha 1 \alpha')_2$, $k = (\alpha 0 \alpha'')_2$, where α, α', and α'' are strings of 0s and 1s with $|\alpha'| = |\alpha''|$. Assume by symmetry that $x = (\beta 1 \beta')_2$ and $y = (\gamma 0 \gamma')_2$, where $|\alpha| = |\beta| = |\gamma|$. Then $k \oplus y = (\beta 0 \gamma'')_2$ is less than x. Hence $k \oplus y \in S$, and $k = (k \oplus y) \oplus y \in S \oplus y$. [See R. P. Sprague, *Tôhoku Math. J.* **41** (1936), 438–444; P. M. Grundy, *Eureka* **2** (1939), 6–8.]

9. The Sprague–Grundy theorem in the previous exercise shows that two piles of x and y sticks are equivalent in play to a single pile of $x \oplus y$ sticks. (There is a nonnegative integer $k < x \oplus y$ if and only if there either is a nonnegative $i < x$ with $i \oplus y < x \oplus y$ or a nonnegative $j < y$ with $x \oplus j < x \oplus y$.) So the k piles are equivalent to a single pile of size $a_1 \oplus \cdots \oplus a_k$. [See C. L. Bouton, *Annals of Math.* (2) **3** (1901–1902), 35–39.]

10. For clarity and brevity we shall write simply xy for $x \otimes y$ and $x + y$ for $x \oplus y$, *in parts (i) through (iv) of this answer only.*

(i) Clearly $0y = 0$ and $x + y = y + x$ and $xy = yx$. Also $1y = y$, by induction on y.

(ii) If $x \ne x'$ and $y \ne y'$ then $xy + xy' + x'y + x'y' \ne 0$, because the definition of xy says that $xy' + x'y + x'y' < xy$ when $x' < x$ and $y' < y$. In particular, if $x \ne 0$ and

$y \neq 0$ then $xy \neq 0$. Another consequence is that, if $x = \text{mex}(S)$ and $y = \text{mex}(T)$ for arbitrary finite sets S and T, we have $xy = \text{mex}\{xj + iy + ij \mid i \in S, j \in T\}$.

cancellation law
distributive law
cancellation law
distributive law
recursively

(iii) Consequently, by induction on the (ordinary) sum of x, y, and z, $(x+y)z$ is

$$\text{mex}\{(x+y)z' + (x'+y)z + (x'+y)z', (x+y)z' + (x+y')z + (x+y')z' \mid 0 \le x' < x,\ 0 \le y' < y,\ 0 \le z' < z\},$$

which is $\text{mex}\{xz' + x'z + x'z' + yz, xz + yz' + y'z + y'z'\} = xz + yz$. In particular, there's a cancellation law: If $xz = yz$ then $(x+y)z = 0$, so $x = y$ or $z = 0$.

(iv) By a similar induction, $(xy)z = \text{mex}\{(xy)z' + (xy' + x'y + x'y')(z + z')\} = \text{mex}\{(xy)z' + (xy')z + (xy')z' + \cdots\} = \text{mex}\{x(yz') + x(y'z) + x(y'z') + \cdots\} = \text{mex}\{(x+x')(yz' + y'z + y'z') + x'(yz)\} = x(yz)$.

(v) If $0 \le x, y < 2^{2^n}$ we shall prove that $x \otimes y < 2^{2^n}$, $2^{2^n} \otimes y = 2^{2^n}y$, and $2^{2^n} \otimes 2^{2^n} = \frac{3}{2}2^{2^n}$. By the distributive law (iii) it suffices to consider the case $x = 2^a$ and $y = 2^b$ for $0 \le a, b < 2^n$. Let $a = 2^p + a'$ and $b = 2^q + b'$, where $0 \le a' < 2^p$ and $0 \le b' < 2^q$; then $x = 2^{2^p} \otimes 2^{a'}$ and $y = 2^{2^q} \otimes 2^{b'}$, by induction on n.

If $p < n-1$ and $q < n-1$ we've already proved that $x \otimes y < 2^{2^{n-1}}$. If $p < q = n-1$, then $x \otimes 2^{b'} < 2^{2^q}$, hence $x \otimes y < 2^{2^n}$. And if $p = q = n-1$, we have $x \otimes y = 2^{2^p} \otimes 2^{2^p} \otimes 2^{a'} \otimes 2^{b'} = (\frac{3}{2}2^{2^p}) \otimes z$, where $z < 2^{2^p}$. Thus $x \otimes y < 2^{2^n}$ in all cases.

By the cancellation law, the nonnegative integers less than 2^{2^n} form a subfield. Hence in the formula

$$2^{2^n} \otimes y \;=\; \text{mex}\{2^{2^n}y' \oplus x'(y \oplus y') \mid 0 \le x' < 2^{2^n}, 0 \le y' < y\}$$

we can choose x' for each y' to exclude all numbers between $2^{2^n}y'$ and $2^{2^n}(y'+1) - 1$; but $2^{2^n}y$ is never excluded.

Finally in $2^{2^n} \otimes 2^{2^n} = \text{mex}\{2^{2^n}(x' \oplus y') \oplus (x' \otimes y') \mid 0 \le x', y' < 2^{2^n}\}$, choosing $x' = y'$ will exclude all numbers up to and including $2^{2^n} - 1$, since $x \otimes x = y \otimes y$ implies that $(x \oplus y) \otimes (x \oplus y) = 0$, hence $x = y$. Choosing $x' = y' \oplus 1$ excludes numbers from 2^{2^n} to $\frac{3}{2}2^{2^n} - 1$, since $(x \otimes x) \oplus x = (y \otimes y) \oplus y$ implies that $x = y$ or $x = y \oplus 1$, and since the most significant bit of $x \otimes x$ is the same as that of x. This same observation shows that $\frac{3}{2}2^{2^n}$ is *not* excluded. QED.

Consider, for example, the subfield $\{0, 1, \ldots, 15\}$. By the distributive law we can reduce $x \otimes y$ to a sum of $x \otimes 1$, $x \otimes 2$, $x \otimes 4$, and/or $x \otimes 8$. We have $2 \otimes 2 = 3$, $2 \otimes 4 = 8$, $4 \otimes 4 = 6$; and multiplication by 8 can be done by multiplying first by 2 and then by 4 or vice versa, because $8 = 2 \otimes 4$. Thus $2 \otimes 8 = 12$, $4 \otimes 8 = 11$, $8 \otimes 8 = 13$.

In general, for $n > 0$, let $n = 2^m + r$ where $0 \le r < 2^n$. There is a $2^{m+1} \times 2^{m+1}$ matrix Q_n such that multiplication by 2^n is equivalent to applying Q_n to blocks of 2^{m+1} bits and working mod 2. For example, $Q_1 = \binom{1\,1}{1\,0}$, and $(\ldots x_4x_3x_2x_1x_0)_2 \otimes 2^1 = (\ldots y_4y_3y_2y_1y_0)_2$, where $y_0 = x_1$, $y_1 = x_1 \oplus x_0$, $y_2 = x_3$, $y_3 = x_3 \oplus x_2$, $y_4 = x_5$, etc. The matrices are formed recursively as follows: Let $Q_0 = R_0 = (1)$ and

$$Q_{2^m+r} = \begin{pmatrix} I & R_m \\ I & 0 \end{pmatrix} \begin{pmatrix} Q_r & & 0 \\ & \ddots & \\ 0 & & Q_r \end{pmatrix}, \qquad R_{m+1} = \begin{pmatrix} R_m & R_m^2 \\ R_m & 0 \end{pmatrix} = Q_{2^{m+1}-1},$$

where Q_r is replicated enough times to make 2^{m+1} rows and columns. For example,

$$Q_2 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}; \qquad Q_3 = Q_2 \begin{pmatrix} Q_1 & 0 \\ 0 & Q_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = R_2.$$

If register x holds any 64-bit number, and if $1 \le j \le 7$, the MMIX instruction `MXOR y,q_j,x` will compute $\mathtt{y} = \mathtt{x} \otimes 2^j$, given the hexadecimal matrix constants

$$\mathtt{q}_1 = \mathtt{c08030200c080302},\quad \mathtt{q}_2 = \mathtt{b06080400b060804},\quad \mathtt{q}_3 = \mathtt{d0b0c0800d0b0c08},$$
$$\mathtt{q}_4 = \mathtt{8d4b2c1880402010},\quad \mathtt{q}_5 = \mathtt{c68d342cc0803020},$$
$$\mathtt{q}_6 = \mathtt{b9678d4bb0608040},\quad \mathtt{q}_7 = \mathtt{deb9c68dd0b0c080}.$$

[For further information, see J. H. Conway, *On Numbers and Games* (1976), Chapter 6, where it is shown that these definitions actually yield an algebraically closed field over the ordinal numbers.]

11. Let $m = 2^{a_s} + \cdots + 2^{a_1}$ with $a_s > \cdots > a_1 \ge 0$ and $n = 2^{b_t} + \cdots + 2^{b_1}$ with $b_t > \cdots > b_1 \ge 0$. Then $m \otimes n = mn$ if and only if $(a_s \mid \cdots \mid a_1) \mathbin{\&} (b_t \mid \cdots \mid b_1) = 0$.

12. If $x = 2^{2^n}a + b$ where $0 \le a, b < 2^{2^n}$, let $x' = x \otimes (x \oplus a)$. Then

$$x' = ((2^{2^n} \otimes a) \oplus b) \otimes ((2^{2^n} \otimes a) \oplus a \oplus b) \;=\; (2^{2^n-1} \otimes a \otimes a) \oplus (b \otimes (a \oplus b)) \;<\; 2^{2^n}.$$

To nim-divide by x we can therefore nim-divide by x' and multiply by $x \oplus a$. [This algorithm is due to H. W. Lenstra, Jr.; see *Séminaire de Théorie des Nombres* (Université de Bordeaux, 1977–1978), exposé 11, exercise 5.]

13. If $a_2 \oplus \cdots \oplus a_k = a_1 \oplus a_3 \oplus \cdots \oplus ((k-2) \otimes a_k) = 0$, every move breaks this condition; we can't have $(a \otimes x) \oplus (b \otimes y) = (a \otimes x') \oplus (b \otimes y')$ when $a \ne b$ unless $(x, y) = (x', y')$.

Conversely, if $a_2 \oplus \cdots \oplus a_k \ne 0$ we can reduce some a_j with $j \ge 2$ to make this sum zero; then a_1 can be set to $a_3 \oplus \cdots \oplus ((k-2) \otimes a_k)$. If $a_2 \oplus \cdots \oplus a_k = 0$ and $a_1 \ne a_3 \oplus \cdots \oplus ((k-2) \otimes a_k)$, we simply reduce a_1 if it is too large. Otherwise there's a $j \ge 3$ such that equality will occur if $(j-2) \otimes a_j$ is replaced by an appropriate smaller value $((j-2) \otimes a_j') \oplus ((i-2) \otimes (a_j \oplus a_j'))$, for some $2 \le i < j$ and $0 \le a_j' < a_j$, because of the definition of nim multiplication; hence both of the desired equalities are achieved by setting $a_j \leftarrow a_j'$ and $a_i \leftarrow a_i \oplus a_j \oplus a_j'$. [This game was introduced in *Winning Ways* by Berlekamp, Conway, and Guy, at the end of Chapter 14.]

14. (a) Each $y = (\ldots y_2 y_1 y_0)_2 = x^T$ determines $x = (\ldots x_2 x_1 x_0)_2$ uniquely, since $x_0 = y_0 \oplus t$ and $\lfloor y/2 \rfloor = \lfloor x/2 \rfloor^{T_{x_0}}$.

(b) When $k > 0$, it is a branching function with labels $t_{\alpha a \beta} = a$ for $|\beta| = k-1$, and $t_\alpha = 0$ for $|\alpha| < k$. But when $k \le 0$, the mapping is not a permutation; in fact, it sends infinitely many 2-adic integers into 0.

[The case $k = 1$ is particularly interesting: Then x^T takes nonnegative integers into nonnegative integers of even parity, negative integers into nonnegative integers of odd parity, and $-1/3 \mapsto -1$. Furthermore $\lfloor x^T/2 \rfloor$ is "Gray binary code," 7.2.1.1–(9).]

(c) If $\rho(x \oplus y) = k$ we have $T(x) \equiv T(y)$ and $x \equiv y + 2^k$ (modulo 2^{k+1}). Hence $\rho(x^T \oplus y^T) = \rho(x \oplus y \oplus T(x) \oplus T(y)) = k$. Conversely, if $\rho(x^T \oplus y^T) = k$ whenever $y = x + 2^k$, we obtain a suitable bit labeling by letting $t_\alpha = (x^T \gg |\alpha|) \bmod 2$ when $x = (\alpha^R)_2$.

(d) This statement follows immediately from (a) and (c). For if we always have $\rho(x \oplus y) = \rho(x^U \oplus y^U) = \rho(x^V \oplus y^V)$, then $\rho(x \oplus y) = \rho(x^U \oplus y^U) = \rho(x^{UV} \oplus y^{UV})$. And if $x^{TU} = x$ for all x, $\rho(x^U \oplus y^U) = \rho(x \oplus y)$ is equivalent to $\rho(x \oplus y) = \rho(x^T \oplus y^T)$.

We can also construct the labelings explicitly: If $W = UV$, note that when $a, b, c \in \{0, 1\}$ we have $W_a = U_a V_{a'}$, $W_{ab} = U_{ab} V_{a'b'}$, and $W_{abc} = U_{abc} V_{a'b'c'}$, where $a' = a \oplus u$, $b' = b \oplus u_a$, $c' = c \oplus u_{ab}$, and so on; hence $w = u \oplus v$, $w_a = u_a \oplus v_{a'}$, $w_{ab} = u_{ab} \oplus v_{a'b'}$, etc. The labeling T inverse to U is obtained by swapping left and right subtrees of all nodes labeled 1; thus $t = u$, $t_a = u_{a'}$, $t_{ab} = u_{a'b'}$, etc.

(e) The explicit constructions in (d) demonstrate that the balance condition is preserved by compositions and inverses, because $\{0', 1'\} = \{0, 1\}$ at each level.

MMIX
MXOR
Conway
ordinal numbers
Lenstra
nim multiplication
Berlekamp
Conway
Guy
parity
Gray binary code

Notes: Hendrik Lenstra observes that branching functions can profitably be viewed as the *isometries* (distance-preserving permutations) of the 2-adic integers, when we use the formula $1/2^{\rho(x\oplus y)}$ to define the "distance" between 2-adic integers x and y. Moreover, the branching functions mod 2^d turn out to be the Sylow 2-subgroup of the group of all permutations of $\{0,1,\ldots,2^{d-1}\}$, namely the unique (up to isomorphism) subgroup that has maximum even order among all subgroups of that group. They also are equivalent to the automorphisms of the complete binary tree with 2^d leaves.

15. Equivalently, $(x+2a)\oplus b=(x\oplus b)+2a$; so we might as well find all b and c such that $(x\oplus b)+c=(x+c)\oplus b$. By (89), the latter is equivalent to $(x\oplus b\oplus c)+2((x\oplus b)\mathbin{\&}c)=(x\oplus c\oplus b)+2(x\mathbin{\&}c)$, so the condition $b\mathbin{\&}c=0$ is necessary and sufficient. Thus the condition $(a\ll 1)\mathbin{\&}b=0$ is necessary and sufficient for the original problem.

16. (a) If $\rho(x\oplus y)=k$ we have $x\equiv y+2^k$ (modulo 2^{k+1}); hence $x+a\equiv y+a+2^k$ and $\rho((x+a)\oplus(y+a))=k$. And $\rho((x\oplus b)\oplus(y\oplus b))$ is obviously k.

(b) The hinted labeling, call it $P(c)$, has 1s on the path corresponding to c, and 0s elsewhere; thus it is balanced. The general animating function can be written

$$x^{P(c_0)^{-a_1}P(c_1)^{-a_2}\ldots P(c_{m-1})^{-a_m}}\oplus c_m,\qquad \text{where } c_j=b_1\oplus\cdots\oplus b_j;$$

so it is balanced if and only if $c_m=0$.

[Incidentally, the set $S=\{P(0)\}\cup\{P(k)\oplus P(k+2^e)\mid k\ge 0 \text{ and } 2^e>k\}$ provides an interesting *basis* for all possible balanced labelings: A labeling is balanced if and only if it is $\bigoplus\{q\mid q\in Q\}$ for some $Q\subseteq S$. This exclusive or is well defined even though Q might be infinite, because only finitely many 1s appear at each node.]

(c) The function $P(c)$ in (b) has this form, because $x^{P(c)}=x\oplus\lfloor x\oplus c\rceil$. Its inverse, $x^{S(c)}=((x\oplus c)+1)\oplus c$, is $x\oplus\lfloor x\oplus\bar c\rceil=x^{P(\bar c)}$. Furthermore we have $x^{P(c)P(d)}=x^{P(c)}\oplus\lfloor x^{P(c)}\oplus d\rceil=x\oplus\lfloor x\oplus c\rceil\oplus\lfloor x\oplus d^{S(c)}\rceil$, because $\lfloor x\oplus y\rceil=\lfloor x^T\oplus y^T\rceil$ for any branching function x^T. Similarly $x^{P(c)P(d)P(e)}=x\oplus\lfloor x\oplus c\rceil\oplus\lfloor x\oplus d^{S(c)}\rceil\oplus\lfloor x\oplus e^{S(d)S(c)}\rceil$, etc. After discarding equal terms we obtain the desired form. The resulting numbers p_j are unique because they are the only values of x at which the function changes sign.

(d) We have, for example, $x\oplus\lfloor x\oplus a\rceil\oplus\lfloor x\oplus b\rceil\oplus\lfloor x\oplus c\rceil=x^{P(a')P(b')P(c')}$ where $a'=a$, $b'=b^{P(a')}$, and $c'=b^{P(a')P(b')}$.

[The theory of animating functions was developed by J. H. Conway in Chapter 13 of his book *On Numbers and Games* (1976), inspired by previous work of C. P. Welter in *Indagationes Math.* **14** (1952), 304–314; **16** (1954), 194–200.]

17. (Solution by M. Slanina.) Such equations are decidable even if we also allow operations such as $x\mathbin{\&}y$, $\bar x$, $x\ll 1$, $x\gg 1$, $2^{\rho x}$, and $2^{\lambda x}$, and even if we allow Boolean combinations of statements and quantifications over integer variables, by translating them into formulas of second-order monadic logic with one successor (S1S). Each 2-adic variable $x=(\ldots x_2x_1x_0)_2$ corresponds to an S1S set variable X, where $j\in X$ means $x_j=1$:

$$\begin{aligned}
z=\bar x \quad&\text{becomes}\quad \forall t(t\in Z\Leftrightarrow t\notin X);\\
z=x\mathbin{\&}y \quad&\text{becomes}\quad \forall t(t\in Z\Leftrightarrow (t\in X\wedge t\in Y));\\
z=2^{\rho x} \quad&\text{becomes}\quad \forall t(t\in Z\Leftrightarrow (t\in X\wedge \forall s(s<t\Rightarrow s\notin X)));\\
z=x+y \quad&\text{becomes}\quad \exists C\,\forall t(0\notin C\ \wedge\ (t\in Z\Leftrightarrow (t{\in}X)\oplus(t{\in}Y)\oplus(t{\in}C))\\
&\qquad\qquad\wedge\ (t{+}1\in C\Leftrightarrow\langle(t{\in}X)(t{\in}Y)(t{\in}C)\rangle)).
\end{aligned}$$

An identity such as $x\mathbin{\&}(-x)=2^{\rho x}$ is equivalent to the translation of

$$\forall X\forall Y\forall Z((\text{integer}(X)\ \wedge\ 0=x+y\ \wedge\ z=x\mathbin{\&}y)\ \Rightarrow\ z=2^{\rho x}),$$

Lenstra
isometries
distance
2-adic integers as a metric space
Sylow 2-subgroup
symmetric group
complete binary tree
infinite exclusive or
Conway
Welter
Slanina
quantifications
second-order monadic logic with one successor
S1S

where integer(X) stands for $\exists t \forall s(s > t \Rightarrow (s \in X \Leftrightarrow t \in X))$. We can include rational 2-adic constants as well; for example, $z = \mu_0$ is equivalent to $0 \in Z \wedge \forall t(t \in Z \Leftrightarrow t+1 \notin Z)$. But of course we cannot include arbitrary (uncomputable) constants.

J. R. Büchi proved that all formulas of S1S are decidable, in *Logic, Methodology, and Philosophy of Science: Proceedings* (Stanford, 1960), 1–11. If we restrict attention to equations, one can show in fact that exponential time suffices.

On the other hand M. Hamburg has shown that the problem would be unsolvable if ρx, λx, or $1 \ll x$ were added to the repertoire; multiplication could then be encoded.

Incidentally, many nontrivial identities exist, even if we use only the operations $x \oplus y$ and $x+1$. For example, C. P. Welter noticed in 1952 that

$$((x \oplus (y+1)) + 1) \oplus (x+1) \;=\; ((((x+1) \oplus y) + 1) \oplus x) + 1.$$

18. Of course row x is entirely blank when x is a multiple of 64. The fine details of this image are apparently "chaotic" and complex, but there is a fairly easy way to understand what happens near the points where the straight lines $x = 64\sqrt{j}$ intersect the hyperbolas $xy = 2^{11}k$, for integers $j, k \geq 1$ that aren't too large.

Indeed, when x and y are integers, the value of $x^2y \gg 11$ is odd if and only if $x^2y/2^{12} \bmod 1 \geq \frac{1}{2}$. Thus, if $x = 64\sqrt{j} + \delta$ and $xy = 2^{11}(k+\epsilon)$ we have

$$\frac{x^2y}{2^{12}} \bmod 1 = \Bigl(\frac{128\sqrt{j}\delta + \delta^2}{4096}\Bigr) y \bmod 1 = \Bigl(\frac{2\delta x - \delta^2}{4096}\Bigr) y \bmod 1 = \Bigl((k+\epsilon)\delta - \frac{\delta^2 y}{4096}\Bigr) \bmod 1,$$

and this quantity has a known relation to $\frac{1}{2}$ when, say, δ is close to a small integer. [See C. A. Pickover and A. Lakhtakia, *J. Recreational Math.* **21** (1989), 166–169.]

19. (a) When $n = 1$, $f(A,B,C)$ has the same value under all arrangements except when $a_0 \neq a_1$, $b_0 \neq b_1$, and $c_0 \neq c_1$; and then it cannot exceed 1. For larger values of n we argue by induction, assuming that $n = 3$ in order to avoid cumbersome notation. Let $A_0 = (a_0, a_1, a_2, a_3)$, $A_1 = (a_4, a_5, a_6, a_7)$, $\ldots$, $C_1 = (c_4, c_5, c_6, c_7)$. Then $f(A,B,C) = \sum_{j \oplus k \oplus l = 0} f(A_j, B_k, C_l) \leq \sum_{j \oplus k \oplus l = 0} f(A_j^*, B_k^*, C_l^*)$ by induction. Thus we can assume that $a_0 \geq a_1 \geq a_2 \geq a_3$, $a_4 \geq a_5 \geq a_6 \geq a_7$, $\ldots$, $c_4 \geq c_5 \geq c_6 \geq c_7$. We can also sort the subvectors $A_0' = (a_0, a_1, a_4, a_5)$, $A_1' = (a_2, a_3, a_6, a_7)$, $\ldots$, $C_1' = (c_2, c_3, c_6, c_7)$ in a similar way. Finally, we can sort $A_0'' = (a_0, a_1, a_6, a_7)$, $A_1'' = (a_2, a_3, a_4, a_5)$, $\ldots$, $C_1'' = (c_2, c_3, c_4, c_5)$, because in each term $a_j b_k c_l$ the number of subscripts $\{j, k, l\}$ with leading bits 01, 10, and 11 must satisfy $s_{01} \equiv s_{10} \equiv s_{11}$ (modulo 2). And these three sorting operations leave A, B, C fully sorted, by exercise 5.3.4–48.

(b) Suppose $A = A^*$, $B = B^*$, and $C = C^*$. Then we have $a_j = \sum_{t=0}^{2^n-1} \alpha_t [j \leq t]$, where $\alpha_j = a_j - a_{j+1} \geq 0$ and we set $a_{2^n} = 0$; similar formulas hold for b_k and c_l. Let $A_{(p)}$ denote the vector $(a_{p(0)}, \ldots, a_{p(2^n-1)})$ when p is a permutation of $\{0, 1, \ldots, 2^n-1\}$. Then by part (a) we have

$$\begin{aligned} f(A_{(p)}, B_{(q)}, C_{(r)}) &= \textstyle\sum_{j \oplus k \oplus l = 0} \sum_{t,u,v} \alpha_t \beta_u \gamma_v [p(j) \leq t][q(k) \leq u][r(l) \leq v] \\ &\leq \textstyle\sum_{j \oplus k \oplus l = 0} \sum_{t,u,v} \alpha_t \beta_u \gamma_v [j \leq t][k \leq u][l \leq v] = f(A, B, C). \end{aligned}$$

[This proof is due to Hardy, Littlewood, and Pólya, *Inequalities* (1934), §10.3.]

(c) The same proof technique extends to any number of vectors. [R. E. A. C. Paley, *Proc. London Math. Soc.* (2) **34** (1932), 263–279, Theorem 15.]

20. The given steps compute the least integer y greater than x such that $\nu y = \nu x$. They're useful for generating all combinations of n objects, taken m at a time (that is, all m-element subsets of an n-element set, with elements represented by 1 bits).

[This tidbit is Hack 175 in HAKMEM, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo No. 239 (29 February 1972).]

magic
Büchi
Hamburg
unsolvable
Welter
hyperbolas
Pickover
Lakhtakia
sorting
Hardy
Littlewood
Pólya
Paley
HAKMEM
combinations+
set repr

21. Set $t \leftarrow y+1$, $u \leftarrow t \oplus y$, $v \leftarrow t \,\&\, y$, $x \leftarrow v - (v \,\&\, {-v})/(u+1)$. If $y = 2^m - 1$ is the *first* m-combination, these eight operations set x to zero. (The fact that $x = \overline{f(\bar{y})}$ does not seem to yield any shorter scheme.)

22. Sideways addition avoids the division: `SUBU t,x,1; ANDN u,x,t; SADD k,t,x; ADDU v,x,u; XOR t,v,x; ADDU k,k,2; SRU t,t,k; ADDU y,v,t`. But we can actually save a step by judiciously using the constant `mone` $= -1$: `SUBU t,x,1; XOR u,t,x; ADDU y,x,u; SADD k,t,y; ANDN y,y,u; SLU t,mone,k; ORN y,y,t`.

23. (a) $(0\ldots01\ldots1)_2 = 2^m - 1$ and $(0101\ldots01)_2 = (2^{2m}-1)/3$.

(b) This solution uses the 2-adic constant $\mu_0 = (\ldots010101)_2 = -1/3$:

$$t \leftarrow x \oplus \mu_0, \quad u \leftarrow (t{-}1) \oplus t, \quad v \leftarrow x \mid u, \quad w \leftarrow v+1, \quad y \leftarrow w + \frac{v \,\&\, \overline{w}}{\sqrt{u+1}}.$$

If $x = (2^{2m}-1)/3$, the operations produce a strange result because $u = 2^{2m+1}-1$.

(c) `XOR t,x,m0; SUBU u,t,1; XOR u,t,u; OR v,x,u; SADD y,u,m0; ADDU w,v,1; ANDN t,v,w; SRU y,t,y; ADDU y,w,y`. [This exercise was inspired by Jörg Arndt.]

24. It's expedient to "prime the pump" by initializing the array to the state that it should have after all multiples of 3, 5, 7, and 11 have been sieved out. We can combine 3 with 11 and 5 with 7, as suggested by E. Wada:

```
      LOC Data_Segment
qbase GREG @ ;N IS 3584 ;n GREG N ;one GREG 1
Q     OCTA #816d129a64b4cb6e                    Q_0 (little-endian)
      LOC Q+N/16
qtop  GREG @                                    End of the Q table
Init  OCTA #9249249249249249|#4008010020040080  Multiples of 3 or 11 in [129..255]
      OCTA #8421084210842108|#0408102040810204  Multiples of 5 or 7
t IS $255 ;x33 IS $0 ;x35 IS $1 ;j IS $4
      LOC #100
Main  LDOU x33,Init; LDOU x35,Init+8
      LDA j,qbase,8;  SUB j,j,qtop              Prepare to set Q_1.
1H    NOR t,x33,x33; ANDN t,t,x35
      STOU t,qtop,j                             Initialize 64 sieve bits.
      SLU t,x33,2; SRU x33,x33,31; OR x33,x33,tPrepare for the next 64 values.
      SLU t,x35,6; SRU x35,x35,29; OR x35,x35,t
      ADD j,j,8;  PBN j,1B                      Repeat until reaching qtop.  ▮
```

Then we cast out nonprimes p^2, p^2+2p, …, for $p = 13$, 17, …, until $p^2 > N$:

```
p IS $0 ;pp IS $1 ;m IS $2 ;mm IS $3 ;q IS $4 ;s IS $5
      LDOU q,qbase,0; LDA pp,qbase,8
      SET p,13; NEG m,13*13,n; SRU q,q,6        Begin with p = 13.
1H    SR m,m,1                                  m ← ⌊(p^2 − N)/2⌋.
2H    SR mm,m,3;  LDOU s,qtop,mm
      AND t,m,#3f;  SLU t,one,t
      ANDN s,s,t;  STOU s,qtop,mm               Zero out a bit.
      ADD m,m,p;  PBN m,2B                      Advance by p bits.
      SRU q,q,1;  PBNZ q,3F                     Move to next potential prime.
2H    LDOU q,pp,0;  INCL pp,8                   Read in another batch
      OR p,p,#7f;  PBNZ q,3F                      of potential primes.
      ADD p,p,2;  JMP 2B                        Skip past 128 nonprimes.
```

mone
magic mask
SADD
Arndt
Wada
little-endian

```
2H   SRU q,q,1
3H   ADD p,p,2;   PBEV q,2B                Set p ← p + 2 until p is prime.
     MUL m,p,p;   SUB m,m,n;   PBN m,1B    Repeat until p^2 > N. ▮
```

The running time, $1172\mu + 5166v$, is of course much less than the time needed for steps P1–P8 of Program 1.3.2′P, namely $10037\mu + 641543v$ (improved to $10096\mu + 215351v$ in exercise 1.3.2′–14). [See P. Pritchard, *Science of Computer Programming* **9** (1987), 17–35, for several instructive variations. In practice, a program like this one tends to slow down dramatically when the sieve is too big for the computer's cache. Better results are obtained by working with a segmented sieve, which contains bits for numbers between $N_0 + k\delta$ and $N_0 + (k+1)\delta$, as suggested by L. J. Lander and T. R. Parkin, *Math. Comp.* **21** (1967), 483–488; C. Bays and R. H. Hudson, *BIT* **17** (1977), 121–127. Here N_0 can be quite large, but δ is limited by the cache size; calculations are done separately for $k = 0, 1, \ldots$. Segmented sieves have become highly developed; see, for example, T. R. Nicely, *Math. Comp.* **68** (1999), 1311–1315, and the references cited there. The author used such a program in 2006 to discover an unusually large gap of length 1370 between 418032645936712127 and the next larger prime.]

25. $(1 + 1 + 25 + 1 + 1 + 25 + 1 + 1 = 56)$ mm; the worm never sees pages 2–500 of Volume 1 or 1–499 of Volume 4. (Unless the books have been placed in little-endian fashion on the bookshelf; then the answer would be 106 mm.) This classic brain-teaser can be found in Sam Loyd's *Cyclopedia* (New York: 1914), pages 327 and 383.

26. We could multiply by `#aa...ab` instead of dividing by 12 (see exercise 1.3.1′–17); but multiplication is slow too. Instead we can use a scheme that is neither big-endian nor little-endian but *transposed*: Put item k into octabyte $8(k \bmod 2^{20})$, where it is shifted left by $5\lfloor k/2^{20}\rfloor$. Since $k < 12000000$, the amount of shift is always less than 60. The MMIX code to put item k into register \$1 is `AND $0,k,[#fffff]; SLU $0,$0,3; LDOU $1,base,$0; SRU $0,k,20; 4ADDU $0,$0,$0; SRU $1,$1,$0; AND $1,$1,#1f`.

[This solution uses 8 large megabytes (2^{23} bytes). *Any* convenient scheme for converting item numbers to octabyte addresses and shift amounts will work, as long as the same method is used consistently. Of course, just '`LDBU $1,base,k`' would be faster.]

27. (a) $((x-1) \oplus x) + x$. [This exercise is based on an idea of Luther Woodrum, who noticed that $((x-1) \,|\, x) + 1 = (x \mathbin{\&} -x) + x$.]

(b) $(y + x) \,|\, y$, where $y = (x-1) \oplus x$.

(c,d,e) $((y \oplus x) + x) \mathbin{\&} y$, $((y \oplus x) + x) \oplus y$, and $\overline{((y \oplus x) + x)} \mathbin{\&} y$, where $y = x-1$.

(f) $x \oplus$(a); alternatively, $y \oplus (y+1)$, where $y = x \,|\, (x-1)$. [The number $(0^\infty 01^a 11^b)_2$ looks simpler, but it apparently requires *five* operations: $((y+1) \mathbin{\&} \bar{y}) - 1$.]

28. A 1 bit indicates x's rightmost 0 (for example, $(101011)_2 \mapsto (000100)_2$); $-1 \mapsto 0$.

29. $\mu_k = \mu_{k+1} \oplus (\mu_{k+1} \ll 2^k)$ [see *STOC* **6** (1974), 125]. This relation holds also for the constants $\mu_{d,k}$ of (48), when $0 \le k < d$, if we start with $\mu_{d,d} = 2^{2^d} - 1$. (There is, however, no easy way to go from μ_k to μ_{k+1}, unless we use the "zip" operation; see (77).)

30. Append '`CSZ rho,x,64`' to (50), thereby adding $1v$ to its execution time. For (51), we simply need to make sure that $rhotab[0] = 8$.

31. In the first place, his code loops forever when $x = 0$. But even after that bug is patched, his assumption that x is a random integer is highly questionable. In most applications when we want to compute ρx for a nonzero 64-bit number x, a more reasonable assumption would be that each of the outcomes $\{0, 1, \ldots, 63\}$ is equally likely. The average and standard deviation then become 31.5 and ≈ 18.5.

Pritchard
cache
segmented sieve
Lander
Parkin
Bays
Hudson
Nicely
gap
Knuth
little-endian
Loyd
big-endian
little-endian
transposed
large megabytes
Woodrum
zip
CSZ

32. '`NEGU y,x; AND y,x,y; MULU y,debruijn,y; SRU y,y,58; LDB rho,decode,y`' has estimated cost $\mu + 14v$, although multiplication by a power of 2 might well be faster than a typical multiplication. Add $1v$ for the correction in answer 30.

33. In fact, an exhaustive calculation shows that exactly 94727 suitable constants a yield a "perfect hash function" for this problem, 90970 of which also identify the power-of-two cases $y = 2^j$; 90918 of those also distinguish the case $y = 0$. The multiplier #`208b2430c8c82129` is uniquely best, in the sense that it doesn't need to refer to table entries above *decode*[32400] when y is known to be a valid input.

34. Proof: If $\rho x = \rho y$ we have either $x = y = 0$ or $x = \alpha 10^k$ and $y = \beta 10^k$; hence $x \oplus y = (\alpha \oplus \beta)01^k = (x-1) \oplus (y-1)$. If $\rho x > \rho y = k$ we have $(x \oplus y) \bmod 2^{k+2} \neq ((x-1) \oplus (y-1)) \bmod 2^{k+2}$.

35. Let $f(x) = x \oplus 3x$. Clearly $f(2x) = 2f(x)$, and $f(4x+1) = 4f(x) + 2$. We also have $f(4x-1) = 4f(x) + 2$, by exercise 34. The hinted identity follows.

Given n, set $u \leftarrow n \gg 1$, $v \leftarrow u + n$, $t \leftarrow u \oplus v$, $n^+ \leftarrow v \,\&\, t$, and $n^- \leftarrow u \,\&\, t$. Clearly $u = \lfloor n/2 \rfloor$ and $v = \lfloor 3n/2 \rfloor$, so $n^+ - n^- = v - u = n$. And this is Reitwiesner's representation, because $n^+ \mid n^-$ has no consecutive 1s. [H. Prodinger, *Integers* **0** (2000), paper a8, 14 pp. Incidentally we also have $f(-x) = f(x)$.]

36. (i) The commands $x \leftarrow x \oplus (x \ll 1)$, $x \leftarrow x \oplus (x \ll 2)$, $x \leftarrow x \oplus (x \ll 4)$, $x \leftarrow x \oplus (x \ll 8)$, $x \leftarrow x \oplus (x \ll 16)$, $x \leftarrow x \oplus (x \ll 32)$ change x to $x^{\oplus}$. (ii) $x^{\&} = ((x+1) \,\&\, \bar{x}) - 1$.

(See exercises 66, 70, and 117 for applications of $x^{\oplus}$; see also exercise 206.)

37. Insert '`CSZ y,x,half`' after the `FLOTU` in (55), where `half` = #`3fe0000000000000`; note that (55) says '`SR`' (not '`SRU`'). No change is needed to (56), if $lamtab[0] = -1$.

38. '`SRU t,x,1; OR y,x,t; SRU t,y,2; OR y,y,t; SRU t,y,4; OR y,y,t; ...; SRU t,y,32; OR y,y,t; SRU y,y,1; CMPU t,x,0; ADDU y,y,t`' takes $15v$.

39. (Solution by H. S. Warren, Jr.) Let $\sigma(x)$ denote the result of smearing x to the right, as in the first line of (57). Compute $x \,\&\, \sigma((x \gg 1) \,\&\, \bar{x})$.

40. Suppose $\lambda x = \lambda y = k$. If $x = y = 0$, (58) certainly holds, regardless of how we define $\lambda 0$. Otherwise $x = (1\alpha)_2$ and $y = (1\beta)_2$, for some binary strings α and β with $|\alpha| = |\beta| = k$; and $x \oplus y < 2^k \le x \,\&\, y$. On the other hand if $\lambda x < \lambda y = k$, we have $x \oplus y \ge 2^k > x \,\&\, y$. And H. S. Warren, Jr., notes that $\lambda x < \lambda y$ if and only if $x < y \,\&\, \bar{x}$.

41. (a) $\sum_{n=1}^{\infty} (\rho n) z^n = \sum_{k=1}^{\infty} z^{2^k}/(1 - z^{2^k}) = z/(1-z) - \sum_{k=0}^{\infty} z^{2^k}/(1 + z^{2^k})$. The Dirichlet generating function is simpler: $\sum_{n=1}^{\infty} (\rho n)/n^z = \zeta(z)/(2^z - 1)$.

(b) $\sum_{n=1}^{\infty} (\lambda n) z^n = \sum_{k=1}^{\infty} z^{2^k}/(1-z)$.

(c) $\sum_{n=1}^{\infty} (\nu n) z^n = \sum_{k=0}^{\infty} z^{2^k}/((1-z)(1+z^{2^k})) = \sum_{k=0}^{\infty} z^{2^k} \mu_k(z)$, where $\mu_k(z) = (1 + z + \cdots + z^{2^k - 1})/(1 - z^{2^{k+1}})$. (The "magic masks" of (47) correspond to $\mu_k(2)$.)

[See *Automatic Sequences* by J.-P. Allouche and J. Shallit (2003), Chapter 3, for further information about the functions ρ and ν, which they denote by ν_2 and s_2.]

42. $e_1 2^{e_1 - 1} + (e_2 + 2) 2^{e_2 - 1} + \cdots + (e_r + 2r - 2) 2^{e_r - 1}$, by induction on r. [D. E. Knuth, *Proc. IFIP Congress* (1971), **1**, 19–27. The fractal aspects of this sum are illustrated in Figs. 3.1 and 3.2 of the book by Allouche and Shallit.]

43. The straightforward implementation of (63), '`SET nu,0; SET y,x; BZ y,Done; 1H ADD nu,nu,1; SUBU t,y,1; AND y,y,t; PBNZ y,1B`' costs $(5 + 4\nu x)v$; it beats the implementation of (62) when $\nu x < 4$, ties when $\nu x = 4$, and loses when $\nu x > 4$.

But we can save $4v$ from the implementation of (62) if we replace the final multiplication-and-shift by '$y \leftarrow y + (y \gg 8)$, $y \leftarrow y + (y \gg 16)$, $y \leftarrow y + (y \gg 32)$, $\nu \leftarrow y \,\&\,$ #`ff`'. [Of course, MMIX's single instruction '`SADD nu,x,0`' is much better.]

perfect hash function
Prodinger
signed right shift
CSZ
Warren
smearing
Warren
Dirichlet generating function
zeta function
magic masks
Allouche
Shallit
ρ
ν
Knuth
fractal

44. Let this sum be $\nu^{(2)}x$. If we can solve the problem for 2^d-bit numbers, we can solve it for 2^{d+1}-bit numbers, because $\nu^{(2)}(2^d x + x') = \nu^{(2)}x + \nu^{(2)}x' + 2^d\nu x$. Therefore a solution analogous to (62) suggests itself, on a 64-bit machine:

Set $z \leftarrow (x \gg 1) \,\&\, \mu_0$ and $y \leftarrow x - z$.
Set $z \leftarrow ((z + (z \gg 2)) \,\&\, \mu_1) + ((y \,\&\, \bar\mu_1) \gg 1)$ and $y \leftarrow (y \,\&\, \mu_1) + ((y \gg 2) \,\&\, \mu_1)$.
Set $z \leftarrow ((z + (z \gg 4)) \,\&\, \mu_2) + ((y \,\&\, \bar\mu_2) \gg 2)$ and $y \leftarrow (y + (y \gg 4)) \,\&\, \mu_2$.
Finally $\nu^{(2)} \leftarrow (((Az) \bmod 2^{64}) \gg 56) + ((((By) \bmod 2^{64}) \gg 65) \ll 3)$,
where $A = (11111111)_{256}$ and $B = (01234567)_{256}$.

But another approach is better on MMIX, which has sideways addition built in:

```
SADD  nu2,x,m0     SADD  t,x,m2       8ADDU  nu2,t,nu2     SADD t,x,m5
SADD  t,x,m1       4ADDU nu2,t,nu2    SADD   t,x,m4        SLU  t,t,5
2ADDU nu2,t,nu2    SADD  t,x,m3       16ADDU nu2,t,nu2     ADD  nu2,nu2,t  ▮
```

[In general, $\nu^{(2)}x = \sum_k 2^k \nu(x \,\&\, \bar\mu_k)$. See *Dr. Dobb's Journal* **8**, 4 (April 1983), 24–37.]

45. Let $d = (x - y) \,\&\, (y - x)$; test if $d \,\&\, y = d$. [Rokicki found that this idea can be used with node addresses to near-randomize binary search trees or Cartesian trees as if they were treaps, without needing an additional random "priority key" in each node. See *U.S. Patent 6347318* (12 February 2002).]

46. `SADD t,x,m; NXOR y,x,m; CSOD x,t,y;` the mask `m` is `~(1<<i|1<<j)`. (In general, these instructions complement the bits specified by $\bar{\texttt{m}}$ if those bits have odd parity.)

47. $y \leftarrow (x \gg \delta) \,\&\, \theta$, $z \leftarrow (x \,\&\, \theta) \ll \delta$, $x \leftarrow (x \,\&\, m) \mid y \mid z$, where $\overline{m} = \theta \mid (\theta \ll \delta)$.

48. Given δ, there are $s_\delta = \prod_{j=0}^{\delta-1} F_{\lfloor (n+j)/\delta \rfloor + 1}$ different δ-swaps, including the identity permutation. (See exercise 4.5.3–32.) Summing over δ gives $1 + \sum_{\delta=1}^{n-1}(s_\delta - 1)$ altogether.

49. (a) The set $S = \{a_1 d_1 + \cdots + a_m d_m \mid \{a_1, \ldots, a_m\} \subseteq \{-1, 0, +1\}\}$ for displacements $\delta_1, \ldots, \delta_m$ must contain $\{n-1, n-3, \ldots, 1-n\}$, because the kth bit must be exchanged with the $(n+1-2k)$th bit for $1 \le k \le n$. Hence $|S| \ge n$. And S contains at most 3^m numbers, at most $2 \cdot 3^{m-1}$ of which are odd.

(b) Clearly $s(mn) \le s(m) + s(n)$, because we can reverse m fields of n bits each. Thus $s(3^m) \le m$ and $s(2 \cdot 3^m) \le m + 1$. Furthermore the reversal of 3^m bits uses only δ-swaps with even values of δ; the corresponding $(\delta/2)$-swaps prove that we have $s((3^m \pm 1)/2) \le m$. These upper bounds match the lower bounds of (a) when $m > 0$.

(c) The string $\alpha a \beta \theta \psi z \omega$ with $|\alpha| = |\beta| = |\theta| = |\psi| = |\omega| = n$ can be changed to $\omega z \psi \theta \beta a \alpha$ with a $(3n+1)$-swap followed by an $(n+1)$-swap. Then $s(n)$ further swaps reverse all. Hence $s(32) \le s(6) + 2 = 4$, and $s(64) \le 5$. Again, equality holds by (a).

Incidentally, $s(63) = 4$ because $s(7) = s(9) = 2$. The lower bound in (a) turns out to be the exact value of $s(n)$ for $1 \le n \le 22$, except that $s(16) = 4$.

50. Express $n = (t_m \ldots t_1 t_0)_3$ in balanced ternary notation. Let $n_j = (t_m \ldots t_j)_3$ and $\delta_j = 2n_j + t_{j-1}$, so that $n_{j-1} - \delta_j = n_j$ and $2\delta_j - n_{j-1} = n_j + t_j$ for $1 \le j \le m$. Let $E_0 = \{0\}$ and $E_{j+1} = E_j \cup (t_j - E_j)$ for $0 \le j < m$. (Thus, for example, $E_1 = \{0, t_0\}$ and $E_2 = \{0, t_0, t_1, t_1 - t_0\}$.)

Assume by induction on j that δ-swaps for $\delta = \delta_1, \ldots, \delta_j$ have changed the n-bit word $\alpha_1 \ldots \alpha_{3^j}$ to $\alpha_{3^j} \ldots \alpha_1$, where each subword α_k has length $n_j + \varepsilon_k$ for some $\varepsilon_k \in E_j$. If $j < m$, a δ_{j+1}-swap within each subword will preserve this assumption. If $j = m$, each α_k has $|\alpha_k| \le m + 1$, because $\varepsilon \in E_j$ implies $|\varepsilon| \le j$. Therefore 2^k-swaps for $\lfloor \lg m \rfloor \ge k \ge 0$ will reverse them all. (Note that a 2^k-swap on a subword of size t, where $2^k < t \le 2^{k+1}$, reduces it to three subwords of sizes $t - 2^k$, $2^{k+1} - t$, $t - 2^k$.)

MMIX
sideways addition
SADD
2ADDU
4ADDU
8ADDU
16ADDU
Rokicki
randomized data structures
binary search trees
Cartesian trees
treaps
Patent
parity
SADD
NXOR
CSOD
balanced ternary notation

zipping
magic masks
inshuffle
Lenfant
transpositions
perfect shuffle
Steele
matrix transposition

51. (a) If $c = (c_{d-1}\ldots c_0)_2$, we must have $\theta_{d-1} = c_{d-1}\mu_{d,d-1}$. But for $0 \le k < d-1$ we can take $\theta_k = c_k\mu_{d,k} \oplus \hat\theta_k$, where $\hat\theta_k$ is *any* mask $\subseteq \mu_{d,k}$.

(b) Let $\Theta(d,c)$ be the set of all such mask sequences. Clearly $\Theta(1,c) = \{c\}$. When $d > 1$ we will have, recursively,

$$\Theta(d,c) = \bigl\{(\theta_0,\ldots,\theta_{d-2},\theta_{d-1},\hat\theta_{d-2},\ldots,\hat\theta_0) \mid \theta_k = \theta'_{k-1} \ddagger \theta''_{k-1},\ \hat\theta_k = \hat\theta'_{k-1} \ddagger \hat\theta''_{k-1}\bigr\},$$

by "zipping together" two sequences $(\theta'_0,\ldots,\theta'_{d-3},\theta'_{d-2},\hat\theta'_{d-3},\ldots,\hat\theta'_0) \in \Theta(d-1,c')$ and $(\theta''_0,\ldots,\theta''_{d-3},\theta''_{d-2},\hat\theta''_{d-3},\ldots,\hat\theta''_0) \in \Theta(d-1,c'')$ for some appropriate θ_0, $\hat\theta_0$, c', and c''.

When c is odd, the bigraph corresponding to (75) has only one cycle; so $(\theta_0,\hat\theta_0, c',c'')$ is either $(\mu_{d,0},0,\lceil c/2\rceil,\lfloor c/2\rfloor)$ or $(0,\mu_{d,0},\lfloor c/2\rfloor,\lceil c/2\rceil)$. But when c is even, the bigraph has 2^{d-1} double bonds; so $\theta_0 = \hat\theta_0$ is any mask $\subseteq \mu_{d,0}$, and $c' = c'' = c/2$. [Incidentally, $\lg|\Theta(d,c)| = 2^{d-1}(d-1) - \sum_{k=1}^{d-1}(2^k-1)\bigl(2^{k-1} - |2^{k-1} - c \bmod 2^k|\bigr)$.]

In both cases we can therefore let $\hat\theta_{d-2} = \cdots = \hat\theta_0 = 0$ and omit the second half of (71) entirely. Of course in case (b) we would do the cyclic shift directly, instead of using (71) at all. But exercise 58 proves that many other useful permutations, such as selective reversal followed by cyclic shift, can also be handled by (71) with $\hat\theta_k = 0$ for all k. The *inverses* of those permutations can be handled with $\theta_k = 0$ for $0 \le k < d-1$.

52. The following solutions make $\hat\theta_j = 0$ whenever possible. We shall express the θ masks in terms of the μ's, for example by writing $\mu_6 \mathbin{\&} \mu_{6,5}$ instead of stating the requested hexadecimal form #`55555555`; the μ form is shorter and more instructive.

(a) $\theta_k = \mu_5 \mathbin{\&} \mu_{6,k}$ and $\hat\theta_k = (\mu_{k+1} \oplus \mu_{k-1}) \mathbin{\&} \mu_{6,k}$ for $0 \le k < 5$; $\theta_5 = \theta_4$. (Here $\mu_{-1} = 0$. To get the "other" perfect shuffle, $(x_{31}x_{63}\ldots x_1x_{33}x_0x_{32})_2$, let $\hat\theta_0 = \mu_{6,0} \mathbin{\&} \bar\mu_1$.)

(b) $\theta_0 = \theta_3 = \hat\theta_0 = \mu_{6,0} \mathbin{\&} \mu_3$; $\theta_1 = \theta_4 = \hat\theta_1 = \mu_{6,1} \mathbin{\&} \mu_4$; $\theta_2 = \theta_5 = \hat\theta_2 = \mu_{6,2} \mathbin{\&} \mu_5$; $\hat\theta_3 = \hat\theta_4 = 0$. [See J. Lenfant, *IEEE Trans.* **C-27** (1978), 637–647, for a general theory.]

(c) $\theta_0 = \mu_{6,0} \mathbin{\&} \mu_4$; $\theta_1 = \mu_{6,1} \mathbin{\&} \mu_5$; $\theta_2 = \theta_4 = \mu_{6,2} \mathbin{\&} \mu_4$; $\theta_3 = \theta_5 = \mu_{6,3} \mathbin{\&} \mu_5$; $\hat\theta_0 = \mu_{6,0} \mathbin{\&} \mu_2$; $\hat\theta_1 = \mu_{6,1} \mathbin{\&} \mu_3$; $\hat\theta_2 = \hat\theta_0 \oplus \theta_2$; $\hat\theta_3 = \hat\theta_1 \oplus \theta_3$; $\hat\theta_4 = 0$.

(d) $\theta_k = \mu_{6,k} \mathbin{\&} \mu_{5-k}$ for $0 \le k \le 5$; $\hat\theta_k = \theta_k$ for $0 \le k \le 2$; $\hat\theta_3 = \hat\theta_4 = 0$.

53. We can write ψ as a product of $d-t$ transpositions, $(u_1v_1)\ldots(u_{d-t}v_{d-t})$ (see exercise 5.2.2–2). The permutation induced by a single transposition (uv) on the index digits, when $u < v$, corresponds to a $(2^v - 2^u)$-swap with mask $\mu_{d,v} \mathbin{\&} \bar\mu_u$. We should do such a swap for (u_1v_1) first, …, $(u_{d-1}v_{d-1})$ last.

In particular, the perfect shuffle in a 2^d-bit register corresponds to the case where $\psi = (01\ldots(d-1))$ is a one-cycle; so it can be achieved by doing such $(2^v - 2^u)$-swaps for $(u,v) = (0,1)$, …, $(0,d-1)$. For example, when $d = 3$ the two-step procedure is $12345678 \mapsto 13245768 \mapsto 15263748$. [Guy Steele suggests an alternative $(d-1)$-step procedure: We can do a 2^k-swap with mask $\mu_{d,k+1} \mathbin{\&} \bar\mu_k$ for $d-1 > k \ge 0$. When $d = 3$ his method takes $12345678 \mapsto 12563478 \mapsto 15263748$.]

The matrix transposition in exercise 52(b) corresponds to $d = 6$ and $(u,v) = (0,3)$, $(1,4)$, $(2,5)$. These operations are the 7-swap, 14-swap, and 28-swap steps for 8×8 matrix transposition illustrated in the text; they can be done in any order.

For exercise 52(c), use $d = 6$ and $(u,v) = (0,2)$, $(1,3)$, $(0,4)$, $(1,5)$. Exercise 52(d) is as easy as 52(b), with $(u,v) = (0,5)$, $(1,4)$, $(2,3)$.

54. Transposition amounts to reversing the bits of the minor diagonals. Successive elements of those diagonals are $m-1$ apart in the register. Simultaneous reversal of all diagonals corresponds to simultaneous reversal of subwords of sizes $1, \ldots, m$, which can be done with 2^k-swaps for $0 \le k < \lceil \lg m \rceil$ (because such transposition is easy

when m is a power of 2, as illustrated in the text). Here's the procedure for $m = 7$:

Given	6-swap	12-swap	24-swap
00 01 02 03 04 05 06	00 **10** 02 **12** 04 **14** 06	00 10 **20** **30** 04 14 **24**	00 10 20 30 **40** **50** **60**
10 11 12 13 14 15 16	**01** 11 **03** 13 **05** 15 **25**	01 11 **21** **31** 05 15 25	01 11 21 31 **41** **51** **61**
20 21 22 23 24 25 26	20 **30** 22 **32** 24 **16** 26	**02** **12** 22 32 **06** 16 26	02 12 22 32 **42** **52** **62**
30 31 32 33 34 35 36	**21** 31 **23** 33 **43** 35 **45**	**03** **13** 23 33 43 **53** **63**	03 13 23 33 43 53 63
40 41 42 43 44 45 46	40 **50** 42 **34** 44 **36** 46	40 50 **60** 34 44 **54** **64**	**04** **14** **24** 34 44 54 64
50 51 52 53 54 55 56	**41** 51 **61** 53 **63** 55 **65**	41 51 61 **35** **45** 55 65	**05** **15** **25** 35 45 55 65
60 61 62 63 64 65 66	60 **52** 62 **54** 64 **56** 66	**42** 52 62 **36** **46** 56 66	**06** **16** **26** 36 46 56 66

55. Given x and y, first set $x \leftarrow x \,|\, (x \ll 2^k)$ and $y \leftarrow y \,|\, (y \ll 2^k)$ for $2d \le k < 3d$. Then set $x \leftarrow (2^{2d+k} - 2^k)$-swap of x with mask $\mu_{2d+k} \,\&\, \bar{\mu}_k$ and $y \leftarrow (2^{2d+k} - 2^{d+k})$-swap of y with mask $\mu_{2d+k} \& \bar{\mu}_{d+k}$ for $0 \le k < d$. Finally set $z \leftarrow x \& y$, then either $z \leftarrow z \,|\, (z \gg 2^k)$ or $z \leftarrow z \oplus (z \gg 2^k)$ for $2d \le k < 3d$, and $z \leftarrow z \,\&\, (2^{n^2} - 1)$. [The idea is to form two $n \times n \times n$ arrays $x = (x_{000} \ldots x_{(n-1)(n-1)(n-1)})_n$ and $y = (y_{000} \ldots y_{(n-1)(n-1)(n-1)})_n$ with $x_{ijk} = a_{jk}$ and $y_{ijk} = b_{jk}$, then transpose coordinates so that $x_{ijk} = a_{ji}$ and $y_{ijk} = b_{ik}$; now $x \,\&\, y$ does all n^3 bitwise multiplications at once. This method is due to V. R. Pratt and L. J. Stockmeyer, *J. Computer and System Sci.* **12** (1976), 210–213.]

56. Use (71) with $\theta_0 = \hat{\theta}_0 = 0$, $\theta_1 = $ `#0010201122113231`, $\theta_2 = $ `#00080e0400080c06`, $\theta_3 = $ `#00000092008100a2`, $\theta_4 = $ `#0000000000000f16`, $\theta_5 = $ `#0000000003199c26`, $\hat{\theta}_4 = $ `#00000c9f0000901a`, $\hat{\theta}_3 = $ `#003a00b50015002b`, $\hat{\theta}_2 = $ `#000103080c0d0f0c`, and $\hat{\theta}_1 = $ `#0020032033233333`.

57. The two choices for each cycle when $d > 1$ have complementary settings. So we can choose a setting in which at least half of the crossbars are inactive, except in the middle column. (See exercise 5.3.4–55 for more about permutation networks.)

58. (a) Every different setting of the crossbars gives a different permutation, because there is exactly one path from input line i to output line j for all $0 \le i, j < N$. (A network with that property is called a "banyan.") The unique such path carries input i on line $l(i,j,k) = ((i \gg k) \ll k) + (j \bmod 2^k)$ after k swapping steps have been made.

(b) We have $l(i\varphi, i, k) = l(j\varphi, j, k)$ if and only if $i \bmod 2^k = j \bmod 2^k$ and $i\varphi \gg k = j\varphi \gg k$; so $(*)$ is necessary. And it is also sufficient, because a mapping φ that satisfies $(*)$ can always be routed in such a way that $j\varphi$ appears on line $l = l(j\varphi, j, k)$ after k steps: If $k > 1$, $j\varphi$ will appear on line $l(j\varphi, j, k-1)$, which is one of the inputs to l. Condition $(*)$ says that we can route it to l without conflict, even if l is $l(i\varphi, i, k)$.

[In *IEEE Transactions* **C-24** (1975), 1145–1155, Duncan Lawrie proved that condition $(*)$ is necessary and sufficient for an arbitrary *mapping* φ of the set $\{0, 1, \ldots, N-1\}$ into itself, when the crossbar modules are allowed to be general 2×2 mapping modules as in exercise 63. Furthermore the mapping φ might be only partially specified, with $j\varphi = *$ ("wild card" or "don't-care") for some values of j. The proof that appears in the previous paragraph actually demonstrates Lawrie's more general theorem.]

(c) $i \bmod 2^k = j \bmod 2^k$ if and only if $k \le \rho(i \oplus j)$; $i \gg k = j \gg k$ if and only if $k > \lambda(i \oplus j)$; and $i\varphi = j\varphi$ if and only if $i = j$, when φ is a permutation.

(d) $\lambda(i\varphi \oplus j\varphi) < \rho(i \oplus j)$ for all i and j if and only if $\lambda(i\tau\varphi \oplus j\tau\varphi) < \rho(i\tau \oplus j\tau) = \rho(i \oplus j)$ for all i and j, because τ is a permutation. [Note that the notation can be confusing: Bit $j\tau\phi$ appears in bit position j if permutation ϕ is applied first, *then* τ.]

(e) Given $i \ne j$ we must prove that $\lambda(i\varphi\psi \oplus j\varphi\psi) \ge \rho(i \oplus j)$. Case 1, i and j are fixed by both φ and ψ: Then $\lambda(i\varphi\psi \oplus j\varphi\psi) = \lambda(i \oplus j) \ge \rho(i \oplus j)$. Case 2, $i\varphi \ne i$ and $j\psi = j$: Then $\lambda(i\varphi\psi \oplus j\varphi\psi) = \lambda(i\varphi \oplus j\varphi) \ge \rho(i \oplus j)$. Case 3, $i\varphi \ne i$ and $j\psi \ne j$: Then $\lambda(i\varphi\psi \oplus j\varphi\psi) = \lambda(i\varphi \oplus j\psi)$. Let $k = \rho(i \oplus j)$, and suppose $\lambda(i\varphi \oplus j\psi) < k$. Then

Pratt
Stockmeyer
permutation networks
banyan
Lawrie
mapping
don't-care
notation

$i \bmod 2^k = j \bmod 2^k$ and $i\varphi \gg k = j\psi \gg k$. Hence $l(i\varphi, i, k) = l(j\psi, j, k)$, and that line carries both $i\varphi$ and $j\psi$. But those two values cannot be equal.

monus
nu(k) summed
Heckel
Schroeppel
magic mask

59. It is $2^{M_d(a,b)}$, where $M_d(a,b)$ is the number of crossbars that have both endpoints in $[a\,..\,b]$. To count them, let $k = \lambda(a \oplus b)$, $a' = a \bmod 2^k$, and $b' = b \bmod 2^k$; notice that $b - a = 2^k + b' - a'$, and $M_d(a,b) = M_{k+1}(a', 2^k + b')$. Counting the crossbars in the top half and bottom half, plus those that jump between halves, gives $M_{k+1}(a', 2^k + b') = M_k(a', 2^k - 1) + M_k(0, b') + ((b'+1) \dot- a')$. Finally, we have $M_k(0, b') = S(b'+1)$; and $M_k(a', 2^k - 1) = M_k(0, 2^k - 1 - a') = S(2^k - a') = k2^{k-1} - ka' + S(a')$, where $S(n)$ is evaluated in exercise 42.

60. A cycle of length $2l$ corresponds to a pattern $u_0 \leftarrow v_0 \leftrightarrow v_1 \rightarrow u_1 \leftrightarrow u_2 \leftarrow v_2 \leftrightarrow \cdots \leftrightarrow v_{2l-1} \rightarrow u_{2l-1} \leftrightarrow u_{2l}$, where $u_{2l} = u_0$ and '$u \leftarrow v$' or '$v \rightarrow u$' means that the permutation sends u to v, '$x \leftrightarrow y$' means that $x = y \oplus 1$.

We can generate a random permutation as follows: Given u_0, there are $2n$ choices for v_0, then $2n-1$ choices for u_1 only one of which causes $u_2 = u_0$, then $2n-2$ choices for v_2, then $2n-3$ choices for u_3 only one of which closes a cycle, etc.

Consequently the generating function is $G(z) = \prod_{j=1}^{n} \frac{2n-2j+z}{2n-2j+1}$. The expected number of cycles, k, is $G'(1) = H_{2n} - \frac{1}{2}H_n = \frac{1}{2}\ln n + \ln 2 - \frac{1}{2}\gamma + O(n^{-1})$. The mean of 2^k is $G(2) = (2^n n!)^2/(2n)! = \sqrt{\pi n} + O(n^{-1/2})$; and the variance is $G(4) - G(2)^2 = (n + 1 - G(2))G(2) = \sqrt{\pi}\, n^{3/2} + O(n)$.

62. The crossbar settings in $P(2^d)$ can be stored in $(2d-1)2^{d-1} = Nd - \frac{1}{2}N$ bits. To get the inverse permutation proceed from right to left. [See P. Heckel and R. Schroeppel, *Electronic Design* **28**, 8 (12 April 1980), 148–152. Note that *any* way to represent an arbitrary permutation requires at least $\lg N! > Nd - N/\ln 2$ bits of memory; so this representation is nearly optimum, spacewise.]

63. (i) $x = y$. (ii) z must be even. (When z is odd we have $(x \ddagger y) \gg z = (y \gg \lceil z/2 \rceil) \ddagger (x \gg \lfloor z/2 \rfloor)$, even when $z < 0$.) (iii) This identity holds for all w, x, y, and z (and also with any other binary bitwise Boolean operator in place of &).

64. $(((z \mid \mu_0) + (z' \,\&\, \bar\mu_0)) \,\&\, \bar\mu_0) \mid (((z \mid \bar\mu_0) + (z' \,\&\, \mu_0)) \,\&\, \mu_0)$. (See (86).)

65. $xu(x^2) + v(x^2) = xu(x)^2 + v(x)^2$.

66. (a) $v(x) = \bigl(u(x)/(1+x^\delta)\bigr) \bmod x^n$; it's the unique polynomial of degree less than n such that $(1 + x^\delta)v(x) \equiv u(x)$ (modulo x^n). (Equivalently, v is the unique n-bit integer such that $(v \oplus (v \ll \delta)) \bmod 2^n = u$.)

(b) We may as well assume that $n = 64m$, and that $u = (u_{m-1} \ldots u_1 u_0)_{2^{64}}$, $v = (v_{m-1} \ldots v_1 v_0)_{2^{64}}$. Set $c \leftarrow 0$; then, using exercise 36, set $v_j \leftarrow u_j^{\oplus} \oplus (-c)$ and $c \leftarrow v_j \gg 63$ for $j = 0, 1, \ldots, m-1$.

(c) Set $c \leftarrow v_0 \leftarrow u_0$; then $v_j \leftarrow u_j \oplus c$ and $c \leftarrow v_j$, for $j = 1, 2, \ldots, m-1$.

(d) Start with $c \leftarrow 0$ and do the following for $j = 0, 1, \ldots, m-1$: Set $t \leftarrow u_j$, $t \leftarrow t \oplus (t \ll 3)$, $t \leftarrow t \oplus (t \ll 6)$, $t \leftarrow t \oplus (t \ll 12)$, $t \leftarrow t \oplus (t \ll 24)$, $t \leftarrow t \oplus (t \ll 48)$, $v_j \leftarrow t \oplus c$, $c \leftarrow (t \gg 61) \times$ #9249249249249249.

(e) Start with $v \leftarrow u$. Then, for $j = 1, 2, \ldots, m-1$, set $v_j \leftarrow v_j \oplus (v_{j-1} \ll 3)$ and (if $j < m-1$) $v_{j+1} \leftarrow v_{j+1} \oplus (v_{j-1} \gg 61)$.

67. Let $n = 2l - 1$ and $m = n - 2d$. If $\frac{1}{2}n < k < n$ we have $x^{2k} \equiv x^{m+t} + x^t$ (modulo $x^n + x^m + 1$), where $t = 2k - n$ is odd. Consequently, if $v = (v_{n-1} \ldots v_1 v_0)_2$, the number

$$w \;=\; u \oplus \bigl(((u \gg d) \oplus (u \gg 2d) \oplus (u \gg 3d) \oplus \cdots) \,\&\, -2^{l-d}\bigr)$$

turns out to equal $(v_{n-2} \ldots v_3 v_1 v_{n-1} \ldots v_2 v_0)_2$. For example, when $l = 4$ and $d = 2$, the square of $u_6 x^6 + \cdots + u_1 x + u_0$ modulo $(x^7 + x^3 + 1)$ is $u_6 x^5 + u_5 x^3 + (u_6 \oplus u_4)x^1 +$

$(u_5 \oplus u_3)x^6 + (u_6 \oplus u_4 \oplus u_2)x^4 + u_1x^2 + u_0$. To compute v, we therefore do a perfect shuffle, $v = \lfloor w/2^l \rfloor \ddagger (w \bmod 2^l)$. The number w can be calculated by methods like those of the previous exercise. [See R. P. Brent, S. Larvala, and P. Zimmermann, *Math. Comp.* **72** (2003), 1443–1452; **74** (2005), 1001–1002.]

68. `SRU t,x,delta; PUT rM,theta; MUX x,t,x.`

69. Notice that the procedure might fail if we attempt to do the 2^{d-1}-shift first instead of last. The key to proving that a small-shift-first strategy works correctly is to watch the spaces *between* selected bits; we will prove that the lengths of these spaces are multiples of 2^{k+1} after the 2^k-shift.

Consider the infinite string $\chi_k = \ldots 1^{t_4}0^{2^k}1^{t_3}0^{2^k}1^{t_2}0^{2^k}1^{t_1}0^{2^k}1^{t_0}$, which represents the situation where $t_l \ge 0$ items need to move $2^k l$ places to the right. A 2^k-shift with mask $\theta_k = \ldots 0^{t_4}0^{2^k}1^{t_3}1^{2^k}0^{t_2}0^{2^k}1^{t_1}1^{2^k}1^{t_0}$ leaves us with the situation represented by the string $\chi_{k+1} = \ldots 1^{T_2}0^{2^k}0^{2^k}1^{T_1}0^{2^k}0^{2^k}1^{T_0}$, where exactly $T_l = t_{2l} + t_{2l+1}$ items need to move $2^{k+1}l$ places. So the claim holds by induction on k.

70. In the previous answer, notice that the string $\psi_k = \theta_k \oplus (\theta_k \ll 1)$ has a 1 at the right of each subword 0^{2^k} of χ_k, and zeros elsewhere; hence $\theta_k = \psi_k^{\oplus}$ in the notation of exercise 36. Furthermore $\psi_{k+1} = (\psi_k \,\&\, \bar\theta_k) \gg 2^k$. Therefore we can proceed as follows:

Set $\psi \leftarrow \bar\chi$, $k \leftarrow 0$, and repeat the following steps while $\psi \ne 0$: Set $x \leftarrow \psi$, then $x \leftarrow x \oplus (x \gg 2^l)$ for $0 \le l < d$, then $\theta_k \leftarrow x$, $\psi \leftarrow (\psi \,\&\, \bar x) \gg 2^k$, and $k \leftarrow k+1$.

If $k < d$ at the end of this computation, the remaining masks $\theta_k, \ldots, \theta_{d-1}$ are zero and those steps can be omitted from (80). Sometimes this procedure gives nonzero masks θ_k that actually do nothing useful, because $t_1 = t_3 = \cdots = 0$. To avoid such redundancy, change '$\theta_k \leftarrow x$' to '$\theta_k \leftarrow x \,\&\, \bigl(x + (x \,\&\, \psi \,\&\, ((\psi \gg 2^k) \mid 2^{d-k}))\bigr)$'.

[See *compress* in H. S. Warren, Jr., *Hacker's Delight* (Addison–Wesley, 2002), §7–4; also G. L. Steele Jr., *U.S. Patent 6715066* (30 March 2004).]

71. Start with $x \leftarrow y$. Do a (-2^k)-shift of x with mask θ_k, for $k = d-1, \ldots, 1, 0$, using the masks of exercise 70. Finally set $z \leftarrow x$ (or $z \leftarrow x \,\&\, \chi$, if you want a "clean" result).

72. $2^{2^{d-1}}x + y$.

73. Equivalently, d sheep-and-goats operations must be able to transform the word $x^\pi = (x_{(2^d-1)\pi} \ldots x_{1\pi}x_{0\pi})_2$ into $(x_{2^d-1} \ldots x_1x_0)_2$, for any permutation π of $\{0, 1, \ldots, 2^d-1\}$. And this can be done by radix-2 sorting (Algorithm 5.2.5R): First bring the odd numbered bits to the left, then bring the bits j for odd $\lfloor j/2 \rfloor$ left, and so on. For example, when $d = 3$ and $x^\pi = (x_3x_1x_0x_7x_5x_2x_6x_4)_2$, the three operations yield successively $(x_3x_1x_7x_5x_0x_2x_6x_4)_2$, $(x_3x_7x_2x_6x_1x_5x_0x_4)_2$, $(x_7x_6x_5x_4x_3x_2x_1x_0)_2$. [See Z. Shi and R. Lee, *Proc. IEEE Conf. ASAP'00* (IEEE CS Press, 2000), 138–148.]

Historical note: The BESM-6 computer, designed in 1965, implemented half of the sheep-and-goats operation: Its «сборка» ("gather" or "pack") command produced $(z \,\&\, \chi) \cdot\!|\!\cdot \bar\chi$, and its «разборка» command ("scatter" or "unpack") went the other way.

74. If $\left|\sum c_{2l} - \sum c_{2l+1}\right| = 2\Delta > 0$, we must rob Δ from the rich half and give it to the poor. There's a position l in the poor half with $c_l = 0$; otherwise that half would sum to at least 2^{d-1}. A cyclic 1-shift that modifies positions l through $(l+t) \bmod 2^d$ makes $c'_{l+k} = c_{l+k+1}$ for $0 \le k < t$, $c'_{l+t} = c_{l+t+1} - \delta$, $c'_{l+t+1} = \delta$, and $c'_{l+k} = c_{l+k}$ for all other k; here δ can be any desired value in the range $0 \le \delta \le c_{l+t+1}$. (We've treated all subscripts modulo 2^d in these formulas.) So we can use the smallest even t such that $c_{l+1} + c_{l+3} + \cdots + c_{l+t+1} = c_l + c_{l+2} + \cdots + c_{l+t} + \Delta + \delta$ for some $\delta \ge 0$.

(The 1-shift need not be cyclic, if we allow ourselves to shift left instead of right. But the cyclic property may be needed in subsequent steps.)

zipper
Brent
Larvala
Zimmermann
MUX
Warren
compress
Steele Jr.
Patent
Shi
Lee
BESM-6 computer
gather
pack
compress
scatter
unpack

75. Equivalently, given indices $0 \le i_0 < i_1 < \cdots < i_{s-1} < i_s = 2^d$ and $0 = j_0 < j_1 < \cdots < j_{s-1} < j_s = 2^d$, we want to map $(x_{2^d-1} \ldots x_1 x_0)_2 \mapsto (x_{(2^d-1)\varphi} \ldots x_{1\varphi} x_{0\varphi})_2$, where $j\varphi = i_r$ for $j_r \le j < j_{r+1}$ and $0 \le r < s$. If $d = 1$, a mapping module does this.

When $d > 1$, we can set the lefthand crossbars so that they route input i_r to line $i_r \oplus ((i_r + r) \bmod 2)$. If s is even, we recursively ask one of the networks $P(2^{d-1})$ inside $P(2^d)$ to solve the problem for indices $\lfloor \{i_0, i_2, \ldots, i_s\}/2 \rfloor$ and $\lfloor \{j_0, j_2, \ldots, j_s\}/2 \rfloor$, while the other solves it for $\lceil \{i_1, i_3, \ldots, i_{s-1}, 2^d\}/2 \rceil$ and $\lceil \{j_0, j_2, \ldots, j_s\}/2 \rceil$. At the right of $P(2^d)$, one can now check that when $j_r \le j < j_{r+1}$, the mapping module for lines j and $j \oplus 1$ has input i_r on line j if $j \equiv r$ (modulo 2), otherwise i_r is on line $j \oplus 1$. A similar proof works when s is odd.

Notes: This network is a slight improvement over a construction by Yu. P. Ofman, *Trudy Mosk. Mat. Obshchestva* **14** (1965), 186–199. We can implement the corresponding network by substituting a "δ-map" for a δ-swap; instead of (69), we use two masks and do seven operations instead of six: $y \leftarrow x \oplus (x \gg \delta)$, $x \leftarrow x \oplus (y \,\&\, \theta) \oplus ((y \,\&\, \theta') \ll \delta)$. This extension of (71) therefore takes only d additional units of time.

76. When a mapping network realizes a permutation, all of its modules must act as crossbars; hence $G(n) \ge \lg n!$. Ofman proved that $G(n) \le 2.5 n \lg n$, and remarked in a footnote that the constant 2.5 could be improved (without giving any details). We have seen that in fact $G(n) \le 2n \lg n$. Note that $G(3) = 3$.

77. Represent an n-network by $(x_{2^n-1} \ldots x_1 x_0)_2$, where $x_k =$ [the binary representation of k is a possible configuration of 0s and 1s when the network has been applied to all 2^n sequences of 0s and 1s], for $0 \le k < 2^n$. Thus the empty network is represented by $2^{2^n} - 1$, and a sorting network for $n = 3$ is represented by $(10001011)_2$. In general, x represents a sorting network for n elements if and only if it represents an n-network and $\nu x = n + 1$, if and only if $x = 2^0 + 2^1 + 2^3 + 2^7 + \cdots + 2^{2^n-1}$.

If x represents α according to these conventions, the representation of $\alpha[i{:}j]$ is $(x \oplus y) \mid (y \gg (2^{n-i} - 2^{n-j}))$, where $y = x \,\&\, \bar{\mu}_{n-j} \,\&\, \mu_{n-i}$.

[See V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer, *STOC* **6** (1974), 122–126.]

78. If $k \ge \lg(m-1)$ the test is valid, because we always have $x_1 + x_2 + \cdots + x_m \ge x_1 \mid x_2 \mid \cdots \mid x_m$, with equality if and only if the sets are disjoint. Moreover, we have $(x_1 + \cdots + x_m) - (x_1 \mid \cdots \mid x_m) \le (m-1)(2^{n-k-1} + \cdots + 1) < (m-1)2^{n-k} \le 2^n$.

Conversely, if $m \ge 2^k + 2$ and $n > 2k$, the test is invalid. We might have, for example, $x_1 + \cdots + x_m = (2^k + 1)(2^{n-k} - 2^{n-2k-1}) + 2^{n-k-1} = 2^n + (2^{n-k} - 2^{n-2k-1})$.

But if $n \le 2k$ the test is still valid when $m = 2^k + 2$, because our proof shows that $x_1 + \cdots + x_m - (x_1 \mid \cdots \mid x_m) \le (2^k + 1)(2^{n-k} - 1) < 2^n$ in that case.

79. $x_\prime = (x - 1) \,\&\, \chi$. (And the formula $x_\prime = ((x - b - 1) \,\&\, a) + b$ corresponds to (85).) These recipes for x' and $x_\prime$ are part of Jörg Arndt's "bit wizardry" routines (2001); Arndt credits (84) to Glenn C. Rhoads.

80. Perhaps the nicest way is to start with $x \leftarrow \chi - 1$ as a signed number; then while $x \ge 0$, set $x \leftarrow x \,\&\, \chi$, visit x, and set $x \leftarrow 2x - \chi$. (The operation $2x - \chi$ can in fact be performed with a single `MMIX` instruction, '`2ADDU x,x,minuschi`'.)

But that trick fails if χ is so large as to be *already* "negative." A slightly slower but more general method starts with $x \leftarrow \chi$ and does the following while $x \ne 0$: Set $t \leftarrow x \,\&\, {-x}$, visit $\chi - t$, and set $x \leftarrow x - t$.

81. $((z \,\&\, \chi) - (z' \,\&\, \chi)) \,\&\, \chi$. (One way to verify this formula is to use (18).)

82. Yes, by letting $z = z'$ in (86): $w \mid (z \,\&\, \bar{\chi})$, where $w = (((z \,\&\, \chi) \ll 1) + \bar{\chi}) \,\&\, \chi$.

recursively
Ofman
δ-map
Ofman
magic masks
Pratt
Rabin
Stockmeyer
Arndt
Rhoads
MMIX
2ADDU

83. (The following iteration propagates bits of y to the right, in the gaps of a scattered accumulator t. Auxiliary variables u and v respectively mark the left and right of each gap; they double in size until being wiped out by w.) Set $t \leftarrow z \mathbin{\&} \chi$, $u \leftarrow (\chi \gg 1) \mathbin{\&} \bar{\chi}$, $v \leftarrow ((\chi \ll 1) + 1) \mathbin{\&} \bar{\chi}$, $w \leftarrow 3(u \mathbin{\&} v)$, $u \leftarrow 3u$, $v \leftarrow 3v$, and $k \leftarrow 1$. Then, while $u \neq 0$, do the following steps: $t \leftarrow t \mid ((t \gg k) \mathbin{\&} u)$, $k \leftarrow k \ll 1$, $u \leftarrow u \mathbin{\&} \overline{w}$, $v \leftarrow v \mathbin{\&} \overline{w}$, $w \leftarrow ((v \mathbin{\&} (u \gg 1) \mathbin{\&} \bar{u}) \ll (k+1)) - ((u \mathbin{\&} (v \ll 1) \mathbin{\&} \bar{v}) \gg k)$, $u \leftarrow u + ((u \mathbin{\&} \bar{v}) \gg k)$, $v \leftarrow v + ((v \mathbin{\&} \bar{u}) \ll k)$. Finally return the answer $((t \gg 1) \mathbin{\&} \chi) \mid (z \mathbin{\&} \bar{\chi})$.

84. $z \leftharpoondown \chi = w - (z \mathbin{\&} \chi)$, where $w = (((z \mathbin{\&} \chi) \ll 1) + \bar{\chi}) \mathbin{\&} \chi$ appears in answer 82; $z \rightharpoondown \chi$ is the quantity t computed (with more difficulty) in the answer to exercise 83.

85. (a) If $x = \texttt{LOC}(a[i,j,k])$ is the drum location corresponding to interleaved bits as stated, then $\texttt{LOC}(a[i+1,j,k]) = x \oplus ((x \oplus ((x \mathbin{\&} \chi) - \chi)) \mathbin{\&} \chi)$ and $\texttt{LOC}(a[i-1,j,k]) = x \oplus ((x \oplus ((x \mathbin{\&} \chi) - 1)) \mathbin{\&} \chi)$, where $\chi = (11111)_8$, by (84) and answer 79. The formulas for $\texttt{LOC}(a[i,j \pm 1,k])$ and $\texttt{LOC}(a[i,j,k \pm 1])$ are similar, with masks 2χ and 4χ.

(b) For random access, let's hope there is room for a table of length 32 giving $f[(i_4i_3i_2i_1i_0)_2] = (i_4i_3i_2i_1i_0)_8$. Then $\texttt{LOC}(a[i,j,k]) = (((f[k] \ll 1) + f[j]) \ll 1) + f[i]$. (On a vintage machine, bitwise computation of f would be much worse than table lookup, because register operations used to be as slow as fetches from memory.)

(c) Let p be the location of the page currently in fast memory, and let $z = -128$. When accessing location x, if $x \mathbin{\&} z \neq p$ it is necessary to read 128 words from drum location $x \mathbin{\&} z$ (after saving the current data to drum location p if it has changed); then set $p \leftarrow x \mathbin{\&} z$. [See *J. Royal Stat. Soc.* **B-16** (1954), 53–55. This scheme of array allocation for external storage was devised independently by E. W. Dijkstra, circa 1960, who called it the "zip-fastener" method. It has often been rediscovered, for example in 1966 by G. M. Morton and later by developers of quadtrees; see Hanan Samet, *Applications of Spatial Data Structures* (Addison–Wesley, 1990). Georg Cantor had considered interleaving the digits of decimal fractions in *Crelle* **84** (1878), 242–258, §7; but he observed that this idea does *not* lead to an easy one-to-one correspondence between the unit interval $[0\,..\,1]$ and the unit square $[0\,..\,1] \times [0\,..\,1]$.]

86. If (p',q',r') bits of (i,j,k) are in the part of the address that does not affect the page number, the total number of page faults is $2\big((2^{p-p'}-1)2^{q+r} + (2^{q-q'}-1)2^{p+r} + (2^{r-r'}-1)2^{p+q}\big)$. Hence we want to minimize $2^{-p'} + 2^{-q'} + 2^{-r'}$ over the set $0 \le p' \le p$, $0 \le q' \le q$, $0 \le r' \le r$, $p'+q'+r' = s$. Since $2^a + 2^b > 2^{a-1} + 2^{b+1}$ when a and b are integers with $a > b+1$, the minimum (for all s) occurs when we select bits from right to left cyclically until running out. For example, when $(p,q,r) = (2,6,3)$ the addressing function would be $(j_5j_4j_3k_2j_2k_1j_1i_1k_0j_0i_0)_2$. In particular, Tocher's scheme is optimal.

[But such a mapping is not necessarily best when the page size isn't a power of 2. For example, consider a 16×16 matrix; the addressing function $(j_3i_3i_2i_1i_0j_2j_1j_0)_2$ is better than $(j_3i_3j_2i_2j_1i_1j_0i_0)_2$ for all page sizes from 17 to 62, except for size 32 when they are equally good.]

87. Set $x \leftarrow x \mathbin{\&} {\sim}((x \mathbin{\&} \texttt{"@@@@@@@@"}) \gg 1)$; each byte $(a_7 \ldots a_0)_2$ is thereby changed to $(a_7a_6(a_5 \wedge \bar{a}_6)a_4 \ldots a_0)_2$. The same transformation works also on 30 additional letters in the Latin-1 supplement to ASCII (for example, æ ↦ Æ); but there's one glitch, ÿ ↦ ß.

[Don Woods used this trick in his original program for the game of Adventure (1976), uppercasing the user's input words before looking them up in a dictionary.]

88. Set $z \leftarrow (x \oplus \bar{y}) \mathbin{\&} h$, then $z \leftarrow ((x \mid h) - (y \mathbin{\&} \bar{h})) \oplus z$.

89. $x' \leftarrow x \gg 1$, $y' \leftarrow y \gg 1$, $t \leftarrow (x' \mathbin{\&} (x \mid \bar{y})) \mid (x \mathbin{\&} y')$, $z \leftarrow (x \mathbin{\&} \bar{y} \mathbin{\&} \bar{\mu}_0) \mid (t \mathbin{\&} \mu_0)$. [From the "**`nasty`**" test program for H. G. Dietz and R. J. Fisher's SWARC compiler (1998).]

table lookup
Dijkstra
zip-fastener
Z order, see zip
Morton
quadtrees
Samet
Cantor
analysis of alg
convex optimization
Tocher
Latin-1
Woods
game
Adventure
Dietz
Fisher
SWARC compiler

90. Insert '$z \leftarrow z \mid ((x \oplus y) \,\&\, l)$' either before or after '$z \leftarrow (x \,\&\, y) + z$'. (The ordering makes no difference, because $x+y \equiv x \oplus y$ (modulo 4) when $x+y$ is even. Therefore MMIX can round to odd at no additional cost, using MOR. Rounding to even in the ambiguous cases is more difficult, and with fixed point arithmetic it is not advantageous.)

91. If $\frac{1}{2}[x,y]$ denotes the average as in (88), the desired result is obtained by repeating the following operations seven times, then concluding with $z \leftarrow \frac{1}{2}[x,y]$ once more:

$$z \leftarrow \frac{1}{2}[x,y], \quad t \leftarrow \alpha \,\&\, h, \quad m \leftarrow (t \ll 1) - (t \gg 7),$$
$$x \leftarrow (m \,\&\, z) \mid (\overline{m} \,\&\, x), \quad y \leftarrow (\overline{m} \,\&\, z) \mid (m \,\&\, y), \quad \alpha \leftarrow \alpha \ll 1.$$

Although rounding errors accumulate through eight levels, the resulting absolute error never exceeds 807/255. Moreover, it is ≈ 1.13 if we average over all 256^3 cases, and it is less than 2 with probability $\approx 94.2\%$. If we round to odd as in exercise 90, the maximum and average error are reduced to 616/255 and ≈ 0.58; the probability of error < 2 rises to $\approx 99.9\%$. Therefore the following MMIX code uses such unbiased rounding:

```
x GREG ;y GREG ;z GREG                     ⎧XOR  t,x,y      MOR m,ffhi,alf⎫
alf GREG ;m GREG ;t IS $255                ⎪MOR  z,rodd,t   PUT rM,m      ⎪
ffhi GREG -1<<56     repeat seven times:   ⎨AND  t,x,y      MUX x,z,x     ⎬
l    GREG #0101010101010101                ⎪ADDU z,z,t      MUX y,y,z     ⎪
rodd GREG #4020100804020101                ⎩                SLU alf,alf,1 ⎭
```

after which the first four instructions are repeated again. The total time for eight α-blends $(67v)$ is less than the cost of eight multiplications.

92. We get $z_j = \lceil (x_j + y_j)/2 \rceil$ for each j. (This fact, noticed by H. S. Warren, Jr., follows from the identity $x + y = ((x \mid y) \ll 1) - (x \oplus y)$. See also the next exercise.)

93. $x - y = (x \oplus y) - ((\bar{x} \,\&\, y) \ll 1)$. ("Borrows" instead of "carries.")

94. $(x - l)_j = (x_j - 1 - b_j) \bmod 256$, where b_j is the "borrow" from fields to the right. So t_j is nonzero if and only if $(x_j \ldots x_0)_{256} < (1 \ldots 1)_{256} = (256^{j+1} - 1)/255$. (The answers to the stated questions are therefore "yes" and "no.")

In general if the constant l is allowed to have *any* value $(l_7 \ldots l_1 l_0)_{256}$, operation (90) makes $t_j \neq 0$ if and only if $(x_j \ldots x_0)_{256} < (l_j \ldots l_0)_{256}$ and $x_j < 128$.

95. Use (90): Test if $h \,\&\, \bigl(t(x \oplus ((x \gg 8) + (x \ll 56))) \mid t(x \oplus ((x \gg 16) + (x \ll 48))) \mid t(x \oplus ((x \gg 24) + (x \ll 40))) \mid t(x \oplus ((x \gg 32) + (x \ll 32)))\bigr) = 0$, where $t(x) = (x - l) \,\&\, \bar{x}$. (These 28 steps reduce to 20 if cyclic shift or MOR is available, or 15 with BDIF and MOR.)

96. Suppose $0 \le x, y < 256$, $x_h = \lfloor x/128 \rfloor$, $x_l = x \bmod 128$, $y_h = \lfloor y/128 \rfloor$, $y_l = y \bmod 128$. Then $[x < y] = \langle \bar{x}_h y_h [x_l < y_l] \rangle$; see exercise 7.1.1–106. And $[x_l < y_l] = [y_l + 127 - x_l \ge 128]$. Hence $[x < y] = \lfloor \langle \bar{x} y z \rangle / 128 \rfloor$, where $z = (\bar{x} \,\&\, 127) + (y \,\&\, 127)$.

It follows that $t = h \,\&\, \langle \bar{x} y z \rangle$ has the desired properties, when $z = (\bar{x} \,\&\, \bar{h}) + (y \,\&\, \bar{h})$. This formula can also be written $t = h \,\&\, {\sim}\langle x \bar{y} \bar{z} \rangle$, where $\bar{z} = {\sim}((\bar{x} \,\&\, \bar{h}) + (y \,\&\, \bar{h})) = (x \mid h) - (y \,\&\, \bar{h})$ by (18).

To get a similar test function for $[x_j \le y_j] = 1 - [y_j < x_j]$, we just interchange $x \leftrightarrow y$ and take the complement: $t \leftarrow h \,\&\, {\sim}\langle x \bar{y} z \rangle = h \,\&\, \langle \bar{x} y \bar{z} \rangle$, where $z = (x \,\&\, \bar{h}) + (\bar{y} \,\&\, \bar{h})$.

97. Set $x' \leftarrow x \oplus \texttt{"********"}$, $y' \leftarrow x \oplus y$, $t \leftarrow h \,\&\, \bigl(x \mid ((x \mid h) - l)\bigr) \,\&\, \bigl(y' \mid ((y' \mid h) - l)\bigr)$, $m \leftarrow (t \ll 1) - (t \gg 7)$, $t \leftarrow t \,\&\, \bigl(x' \mid ((x' \mid h) - l)\bigr)$, $z \leftarrow (m \,\&\, \texttt{"********"}) \mid (\overline{m} \,\&\, y)$. (20 steps.)

98. Set $u \leftarrow x \oplus y$, $z \leftarrow (\bar{x} \,\&\, \bar{h}) + (y \,\&\, \bar{h})$, $t \leftarrow h \oplus x \oplus (u \mid (x \oplus z))$, $v \leftarrow ((t \ll 1) - (t \gg 7)) \,\&\, u$, $z \leftarrow x \oplus v$, $w \leftarrow y \oplus v$. [This 14-step procedure invokes answer 96 to compute $t = h \,\&\, \langle \bar{x} y z \rangle$, using the footprint method of Section 7.1.2 to evaluate the median in only

MMIX
MOR
Rounding to even
round to odd
MUX
unbiased rounding
Warren
identity
Borrows
carries
borrow
MOR
BDIF
cyclic shift
medians
footprint method
median

three steps when $x \oplus y$ is known. Of course the MMIX solution is much quicker, if available: `BDIF t,x,y; ADDU z,y,t; SUBU w,x,t`.]

99. In this potpourri, each of the eight bytes appears to be solving a different kind of problem; we must recast the conditions so that they fit into a common framework: $f_0 = [x_0 \oplus \texttt{'!'} \le 0]$, $f_1 = [x_1 \oplus \texttt{'*'} > 0]$, $f_2 = [x_2 \le \texttt{'A'} - 1]$, $f_3 = [x_3 > \texttt{'z'}]$, $f_4 = [x_4 > \texttt{'a'} - 1]$, $f_5 = [x_5 \oplus \texttt{'0'} \le 9]$, $f_6 = [x_6 \oplus 255 > 86]$, $f_7 = [x_7 \oplus \texttt{'?'} \le 3]$. Aha! We can use the formulas in answer 96, adjusting d to switch between $\le$ and $>$ as needed: $a = (\texttt{'?'}(255)\texttt{'0'}00\texttt{'*'}\texttt{'!'})_{256} = \#\texttt{3fff300000002a21}$; $b = \bar{h} = \#\texttt{7f7f7f7f7f7f7f7f}$; $c = \bar{h} \mathbin{\&} {\sim}(3(86)9(\texttt{'a'}-1)\texttt{'z'}(\texttt{'A'}-1)00)_{256} = \#\texttt{7c29761f053f7f7f}$ (the hardest one); $d = \#\texttt{8000800000800080}$; and $e = h = \#\texttt{8080808080808080}$.

100. We want $u_j = x_j + y_j + c_j - 10c_{j+1}$ and $v_j = x_j - y_j - b_j + 10b_{j+1}$, where c_j and b_j are the "carry" and "borrow" into digit position j. Set $u' \leftarrow (x + y + (6\ldots66)_{16}) \bmod 2^{64}$ and $v' \leftarrow (x - y) \bmod 2^{64}$. Then we find $u'_j = x_j + y_j + c_j + 6 - 16c_{j+1}$ and $v'_j = x_j - y_j - b_j + 16b_{j+1}$ for $0 \le j < 16$, by induction on j. Hence u' and v' have the same pattern of carries and borrows as if we were working in radix 10, and we have $u = u' - 6(\bar{c}_{16}\ldots\bar{c}_2\bar{c}_1)_{16}$, $v = v' - 6(b_{16}\ldots b_2 b_1)_{16}$. The following computation schemes therefore provide the desired results (10 operations for addition, 9 for subtraction):

$$y' \leftarrow y + (6\ldots66)_{16},\ u' \leftarrow x + y', \qquad v' \leftarrow x - y,$$
$$t \leftarrow \langle \bar{x}\bar{y}'u' \rangle \mathbin{\&} (8\ldots88)_{16}, \qquad t \leftarrow \langle \bar{x}yv' \rangle \mathbin{\&} (8\ldots88)_{16},$$
$$u \leftarrow u' - t + (t \gg 2); \qquad v \leftarrow v' - t + (t \gg 2).$$

101. For subtraction, set $z \leftarrow x - y$; for addition, set $z \leftarrow x + y + \#\texttt{e8c4c4fc18}$, where this constant is built from $256 - 24 = \#\texttt{e8}$, $256 - 60 = \#\texttt{c4}$, and $65536 - 1000 = \#\texttt{fc18}$. Borrows and carries will occur between fields as if mixed-radix subtraction or addition were being performed. The remaining task is to correct for cases in which borrows occurred or carries did not; we can do this easily by inspecting individual digits, because the radices are less than half of the field sizes: Set $t \leftarrow z \mathbin{\&} \#\texttt{8080808000}$, $t \leftarrow (t \ll 1) - (t \gg 7) - ((t \gg 15) \mathbin{\&} 1)$, $z \leftarrow z - (t \mathbin{\&} \#\texttt{e8c4c4fc18})$. [See Stephen Soule, *CACM* **6** (1975), 344–346. We're lucky that the '`c`' in '`fc18`' is even.]

102. (a) We assume that $x = (x_{15}\ldots x_0)_{16}$ and $y = (y_{15}\ldots y_0)_{16}$, with $0 \le x_j, y_j < 5$; the goal is to compute $u = (u_{15}\ldots u_0)_{16}$ and $v = (v_{15}\ldots v_0)_{16}$, with components $u_j = (x_j + y_j) \bmod 5$ and $v_j = (x_j - y_j) \bmod 5$. Here's how:

$$u \leftarrow x + y, \qquad v \leftarrow x - y + 5l,$$
$$t \leftarrow (u + 3l) \mathbin{\&} h, \qquad t \leftarrow (v + 3l) \mathbin{\&} h,$$
$$u \leftarrow u - ((t - (t \gg 3)) \mathbin{\&} 5l); \qquad v \leftarrow v - ((t - (t \gg 3)) \mathbin{\&} 5l).$$

Here $l = (1\ldots1)_{16} = (2^{64}-1)/15$, $h = 8l$. (Addition in 7 operations, subtraction in 8.)

(b) Now $x = (x_{20}\ldots x_0)_8$, etc., and we must be more careful to confine carries:

$$t \leftarrow x + \bar{h},$$
$$z \leftarrow (t \mathbin{\&} \bar{h}) + (y \mathbin{\&} \bar{h}), \qquad z \leftarrow (x \mid h) - (y \mathbin{\&} \bar{h}),$$
$$t \leftarrow (y \mid z) \mathbin{\&} t \mathbin{\&} h, \qquad t \leftarrow (y \mid \bar{z}) \mathbin{\&} \bar{x} \mathbin{\&} h,$$
$$u \leftarrow x + y - (t + (t \gg 2)); \qquad v \leftarrow x - y + t + (t \gg 2).$$

Here $h = (4\ldots4)_8 = (2^{65} - 4)/7$. (Addition in 11 operations, subtraction in 10.)

Similar procedures work, of course, for other moduli. In fact we can do multibyte arithmetic on the coordinates of toruses in general, with different moduli in each component (see 7.2.1.3–(66)).

MMIX
BDIF
potpourri
carry
borrow
medians
Borrows
carries
Soule
multibyte arithmetic
toruses

103. Let h and l be the constants in (87) and (88). Addition is easy: $u \leftarrow x\,|\,((x\&\bar{h})+y)$. For subtraction, take away 1 and add the complement: $t \leftarrow (x\&\bar{l})\gg 1$, $v \leftarrow t\,|\,(t+(y\oplus l))$.

104. Yes, in 19: Let $a = (((1901 \ll 4) + 1) \ll 5) + 1$, $b = (((2099 \ll 4) + 12) \ll 5) + 28$. Set $m \leftarrow (x \gg 5)\ \&\ \texttt{\#f}$ (the month), $c \leftarrow \texttt{\#10}\ \&\ {\sim}((x \mid (x \gg 1)) \gg 5)$ (the leap year correction), $u \leftarrow b + \texttt{\#3}\ \&\ ((\texttt{\#3bbeecc} + c) \gg (m+m))$ (the *max_day* adjustment), and $t \leftarrow ((x \oplus a \oplus (x-a)) \mid (x \oplus u \oplus (u-x)))\ \&\ \texttt{\#1000220}$ (the test for unwanted carries).

105. Exercise 98 explains how to compute bytewise min and max; a simple modification will compute min in some byte positions and max in others. Thus we can "sort by perfect shuffles" as in Section 5.3.4, Fig. 57, if we can permute bytes between x and y appropriately. And such permutation is easy, by exercise 1. [Of course there are much simpler and faster ways to sort 16 bytes. But see S. Albers and T. Hagerup, *Inf. and Computation* **136** (1997), 25–51, for asymptotic implications of this approach.]

106. The n bits are regarded as g fields of g bits each. First the nonzero fields are detected (t_1), and we form a word y that has $(y_{g-1}\ldots y_0)_2$ in each g-bit field, where $y_j = [\text{field } j \text{ of } x \text{ is nonzero}]$. Then we compare each field with the constants 2^{g-1}, $\ldots$, 2^0 (t_2), and form a mask m that identifies the most significant nonzero field of x. After putting g copies of that field into z, we test z as we tested y (t_3). Finally an appropriate sideways addition of t_2 and t_3 (g-bit-wise) yields λ. (Try the case $g = 4$, $n = 16$.)

To compute 2^λ without shifting left, replace '$t_2 \ll 1$' by '$t_2 + t_2$', and replace the final line by $w \leftarrow (((a \cdot (t_3 \oplus (t_3 \gg g))) \bmod 2^n) \gg (n-g)) \cdot l$; then $w\ \&\ m$ is $2^{\lambda x}$.

107.
```
h  GREG #8000800080008000      CSNZ x,q,z          SUBU t,q,t
ms GREG #00ff0f0f33335555      CSNZ lam,q,t        OR   t,t,y
1H ANDN q,x,m5              2H SLU  q,x,16         AND  t,t,h
   SRU  z,x,32                 ADDU x,x,q       5H SLU  q,t,15
   CSNZ x,q,z                  SLU  q,x,32         ADDU t,t,q
   ZSNZ lam,q,32               ADDU x,x,q          SLU  q,t,30
   ANDN q,x,m4              3H ANDN y,x,ms         ADDU t,t,q
   SRU  z,x,16              4H XOR  t,x,y       6H SRU  q,t,60
   ADD  t,lam,16               OR   q,y,h          ADDU lam,lam,q  ▌
```

The total time $25v$ (and no mems) should be increased by v for a fair comparison with (56), because (56) doesn't clobber x.

108. For example, let e be minimum so that $n \le 2^e \cdot 2^{2^e}$. If n is a multiple of 2^e, we can use 2^e fields of size $n/2^e$, with e reductions in step B1; otherwise we can use 2^e fields of size $2^{\lceil \lg n \rceil - e - 1}$, with $e+1$ reductions in step B1. In either case there are e iterations in steps B2 and B5, so the total running time is $O(e) = O(\log\log n)$.

109. Start with $x \leftarrow x\ \&\ {-x}$ and apply Algorithm B. (Step B4 of that algorithm can be slightly simplified in this special case, using a constant l instead of $x \oplus y$.)

110. Let $s = 2^d$ where $d = 2^e - e$. We will use s-bit fields in n-bit words.

K1. [Stretch $x \bmod s$.] Set $y \leftarrow x\ \&\ (s-1)$. Then set $t \leftarrow y\ \&\ \bar{\mu}_j$ and $y \leftarrow y \oplus t \oplus (t \ll 2^j(s-1))$ for $e > j \ge 0$. Finally set $y \leftarrow (y \ll s) - y$. [If $x = (x_{2^e-1}\ldots x_0)_2$ we now have $y = (y_{2^e-1}\ldots y_0)_{2^s}$, where $y_j = (2^s-1)x_j[j<d]$.]

K2. [Set up minterms.] Set $y \leftarrow y \oplus (a_{2^e-1}\ldots a_0)_{2^s}$, where $a_j = \mu_{d,j}$ for $0 \le j < d$ and $a_j = 2^s - 1$ for $d \le j < 2^e$.

K3. [Compress.] Set $y \leftarrow (y \gg 2^j s)$ for $e > j \ge 0$, then $y \leftarrow y\ \&\ (2^s - 1)$. [Now $y = 1 \ll (x \bmod s)$. This is the key point that makes the algorithm work.]

leap year
table lookup by shifting
bytewise min and max
perfect shuffles
Albers
Hagerup
sideways addition
CSNZ
ZSNZ
Stretch
magic

K4. [Finish.] Set $y \leftarrow y \mid (y \ll 2^j s)$ for $0 \le j < e$. Finally set $y \leftarrow y \mathbin{\&} (\mu_{2^e,j} \oplus -((x \gg j) \mathbin{\&} 1))$ for $d \le j < 2^e$. ▮

111. The n bits are divided into fields of s bits each, although the leftmost field might be shorter. First y is set to flag the all-1 fields. Then $t = (\ldots t_1 t_0)_{2^s}$ contains candidate bits for q, including "false drops" for certain patterns 01^k with $s \le k < r$. We always have $\nu t_j \le 1$, and $t_j \ne 0$ implies $t_{j-1} = 0$. The bits of u and v subdivide t into two parts so that we can safely compute $m = (t \gg 1) \mid (t \gg 2) \mid \cdots \mid (t \gg r)$, before making a final test to eliminate the false drops.

112. Notice that if $q = x \mathbin{\&} (x \ll 1) \mathbin{\&} \cdots \mathbin{\&} (x \ll (r-1)) \mathbin{\&} {\sim}(x \ll r)$ then we have $x \mathbin{\&} \overline{x+q} = x \mathbin{\&} (x \ll 1) \mathbin{\&} \cdots \mathbin{\&} (x \ll (r-1))$.

If we can solve the stated problem in $O(1)$ steps, we can also extract the most significant bit of an r-bit number in $O(1)$ steps: Apply the case $n = 2r$ to the number $2^{2^n} - 1 - x$. Conversely, a solution to the extraction problem can be shown to yield a solution to the $1^r 0$ problem. Exercise 110 therefore implies a solution in $O(\log\log r)$ steps.

113. Let $0' = 0$, $x'_0 = x_0$, and construct $x'_{i'} = x_i$ for $1 \le i \le r$ as follows: If $x_i = a \circ_i b$ and $\circ_i \notin \{+, -, \ll\}$, let $i' = (i-1)' + 1$ and $x'_{i'} = a' \circ_i b'$, where $a' = x'_{j'}$ if $a = x_j$ and $a' = a$ if $a = c_i$. If $x_i = a \ll c$, let $i' = (i-1)' + 2$ and $(x'_{i'-1}, x'_{i'}) = (a' \mathbin{\&} (\lfloor 2^{n-c} \rfloor - 1), x'_{i'-1} \ll c)$. If $x_i = a + b$, let $i' = (i-1)' + 6$ and let $(x'_{(i-1)'+1}, \ldots, x'_{i'})$ compute $((a' \mathbin{\&} \bar{h}) + (b' \mathbin{\&} \bar{h})) \oplus ((a' \oplus b') \mathbin{\&} h)$, where $h = 2^{n-1}$. And if $x_i = a - b$, do the similar computation $((a' \mid h) - (b' \mathbin{\&} \bar{h})) \oplus ((a' \equiv b') \mathbin{\&} h)$. Clearly $r' \le 6r$.

114. Simply let $X_i = X_{j(i)} \circ_i X_{k(i)}$ when $x_i = x_{j(i)} \circ_i x_{k(i)}$, $X_i = C_i \circ_i X_{k(i)}$ when $x_i = c_i \circ_i x_{k(i)}$, and $X_i = X_{j(i)} \circ_i C_i$ when $x_i = x_{j(i)} \circ_i c_i$, where $C_i = c_i$ when c_i is a shift amount, otherwise $C_i = (c_i \ldots c_i)_{2^n} = (2^{mn} - 1)c_i/(2^n - 1)$. This construction is possible thanks to the fact that variable-length shifts are prohibited.

[Notice that if $m = 2^d$, we can use this idea to simulate 2^d instances of $f(x, y_i)$; then $O(d)$ further operations allow "quantification."]

115. (a) $z \leftarrow x \mathbin{\&} (\bar{x} \ll 1) \mathbin{\&} (x \ll 2)$, $y \leftarrow x \mathbin{\&} (x + z)$. [This problem was posed to the author by Vaughan Pratt in 1977.]

(b) First find $x_l \leftarrow (x \ll 1) \mathbin{\&} \bar{x}$ and $x_r \leftarrow x \mathbin{\&} (\bar{x} \ll 1)$, the left and right ends of x's blocks; and set $x'_r = x_r \mathbin{\&} (x_r - 1)$. Then $z_e \leftarrow x'_r \mathbin{\&} (x'_r - (x_l \mathbin{\&} \bar{\mu}_0))$ and $z_o \leftarrow x'_r \mathbin{\&} (x'_r - (x_l \mathbin{\&} \mu_0))$ are the right ends that are followed by a left end in even or odd position, respectively. The answer is $y \leftarrow x \mathbin{\&} (x + (z_e \mathbin{\&} \bar{\mu}_0) + (z_o \mathbin{\&} \mu_0))$; it can be simplified to $y \leftarrow x \mathbin{\&} (x + (z_e \oplus (x'_r \mathbin{\&} \mu_0)))$.

(c) This case is impossible, by Corollary I.

116. The language L is well defined, by Lemma A (except that the presence or absence of the empty string is irrelevant). A language is regular if and only if it can be defined by a finite state automaton, and a 2-adic integer is rational if and only if it can be defined by a finite state automaton that ignores its inputs. The identity function corresponds to the language $L = 1(0 \cup 1)^*$, and a simple construction will define an automaton that corresponds to the sum, difference, or Boolean combination of the numbers defined by any two given automata acting on the sequence $x_0 x_1 x_2 \ldots$. Hence L is regular.

In exercise 115, L is (a) $11^*(000^*1(0 \cup 1)^* \cup 0^*)$; (b) $11^*(00(00)^*1(0 \cup 1)^* \cup 0^*)$.

117. Incidentally, the stated language L corresponds to an inverse Gray binary code: It defines a function with the property that $f(2x) = {\sim}f(2x+1)$, and $g(f(2x)) = g(f(2x+1)) = x$, where $g(x) = x \oplus (x \gg 1)$ (see Eq. 7.2.1.1–(9)).

118. If $x = (x_{n-1} \ldots x_1 x_0)_2$ and $0 \le a_j \le 2^j$ for $0 \le j < n$, we have $\sum_{j=0}^{n-1} a_j x_j = \sum_{j=0}^{n-1} (a_j \dot{-} (\bar{x} \mathbin{\&} 2^j))$. Take $a_j = \lfloor 2^{j-1} \rfloor$ to get $x \gg 1$.

extract the most significant bit
quantification
Pratt
finite state automaton
Gray binary code

Conversely, the following argument by M. S. Paterson proves that monus must be used at least $n-1$ times: Consider any chain for $f(x)$ that uses addition, subtraction, bitwise Booleans, and k occurrences of the "underflow" operation $y \lhd z = (2^n-1)[y<z]$. If $k < n-1$ there must be two n-bit numbers x' and x'' such that $x' \bmod 2 = x'' \bmod 2 = 0$ and such that all k of the $\lhd$'s yield the same result for both x' and x''. Then $f(x') \bmod 2^j = f(x'') \bmod 2^j$ when $j = \rho(x' \oplus x'')$. So $f(x)$ is not the function $x \gg 1$. Paterson underflow

119. $z \leftarrow x \oplus y$, $f \leftarrow 2^p \,\&\, \bar{z} \,\&\, (z-1)$. (See (90).)

120. Generalizing Corollary W, these are the functions such that $f(x_1,\ldots,x_m) \equiv f(y_1,\ldots,y_m)$ (modulo 2^k) whenever $x_j \equiv y_j$ (modulo 2^k) for $1 \le j \le m$, for $0 \le k \le n$. The least significant bit is a binary function of m variables, so it has 2^{2^m} possibilities. The next-to-least is a binary function of $2m$ variables, namely the bits of ($x_1 \bmod 4$, $\ldots, x_m \bmod 4$), so it has $2^{2^{2m}}$; and so on. Thus the answer is $2^{2^m+2^{2m}+\cdots+2^{nm}}$.

121. (a) If f has a period of length pq, where $q>1$ is odd, its p-fold iteration $f^{[p]}$ has a period of length q, say $y_0 \mapsto y_1 \mapsto \cdots \mapsto y_q = y_0$ where $y_{j+1} = f^{[p]}(y_j)$ and $y_1 \ne y_0$. But then, by Corollary W, we must have $y_0 \bmod 2^{n-1} \mapsto y_1 \bmod 2^{n-1} \mapsto \cdots \mapsto y_q \bmod 2^{n-1}$ in the corresponding $(n-1)$-bit chain. Consequently $y_1 \equiv y_0$ (modulo 2^{n-1}), by induction on n. Hence $y_1 = y_0 \oplus 2^{n-1}$, and $y_2 = y_0$, etc., a contradiction.

(b) $x_1 = x_0 + x_0$, $x_2 = x_0 \gg (p-1)$, $x_3 = x_1 \mid x_2$; a period of length p starts with the value $x_0 = (1 + 2^p + 2^{2p} + \cdots) \bmod 2^n$.

122. Subtraction is analogous to addition; Boolean operations are even simpler; and constants have only one bit pattern. The only remaining case is $x_r = x_j \gg c$, where we have $S_r = S_j + c$; the shift goes left when $c<0$. Then $V_{pqr} = V_{(p+c)(q+c)j}$, and

$$x_r \,\&\, \lfloor 2^p - 2^q \rfloor \;=\; \bigl((x_j \,\&\, \lfloor 2^{p+c} - 2^{q+c} \rfloor) \gg c\bigr) \,\&\, (2^n - 1).$$

Hence $|X_{pqr}| \le |X_{(p+c)(q+c)j}| \le B_j = B_r$ by induction.

123. If $x = (x_{g-1}\ldots x_0)_2$, note first that $t = 2^{g-1}(x_0\ldots x_{g-1})_{2^g}$ in (104); hence $y = (x_0\ldots x_{g-1})_2$ as claimed. Theorem P now implies that $\lfloor \frac{1}{3}\lg g \rfloor$ broadword steps are needed to multiply by a_{g+1} and by a_{g-1}. At least one of those multiplications must require $\lfloor \frac{1}{6}\lg g \rfloor$ or more steps.

124. Initially $t \leftarrow 0$, $x_0 = x$, $U_0 = \{1, 2, \ldots, 2^{n-1}\}$, and $1' \leftarrow 0$. When advancing $t \leftarrow t+1$, if the current instruction is $r_i \leftarrow r_j \pm r_k$ we simply define $x_t = x_{j'} \pm x_{k'}$ and $i' \leftarrow t$. The cases $r_i \leftarrow r_j \circ r_k$ and $r_i \leftarrow c$ are similar.

If the current instruction branches when $r_i \le r_j$, define $x_t = x_{t-1}$ and let $V_1 = \{x \in U_{t-1} \mid x_{i'} \le x_{j'}\}$, $V_0 = U_t \setminus V_1$. Let U_t be the larger of V_0 and V_1; branch if $U_t = V_1$. Notice that $|U_t| \ge |U_{t-1}|/2$ in this case.

If the current instruction is $r_i \leftarrow r_j \gg r_k$, let $W = \{x \in U_{t-1} \mid x \& \lfloor 2^{\lg n + s} - 2^s \rfloor \ne 0$ for some $s \in S_{k'}\}$, and note that $|W| \le |S_{k'}| \lg n \le 2^{t-1+e+f}$. Let $V_c = \{x \in U_{t-1} \setminus W \mid x_{k'} = c\}$ for $|c| < n$, and $V_n = U_{t-1} \setminus W \setminus \bigcup_{|c|<n} V_c$. Lemma B tells us that at most $B_{k'} + 1 \le 2^{2^{t-1}-1} + 1$ of the sets V_c are nonempty. Let U_t be the largest; and if it is V_c, define $x_t = x_{j'} \gg c$, $i' \leftarrow t$. In this case $|U_t| \ge (|U_{t-1}| - 2^{t-1+e+f})/(2^{2^{t-1}-1} + 1)$.

Similarly for $r_i \leftarrow M[r_j \bmod 2^m]$ or $M[r_j \bmod 2^m] \leftarrow r_i$, let $W = \{x \in U_{t-1} \mid x \,\&\, \lfloor 2^{m+s} - 2^s \rfloor \ne 0$ for some $s \in S_{j'}\}$, and $V_z = \{x \in U_{t-1} \setminus W \mid x_{j'} \bmod 2^m = z\}$, for $0 \le z < 2^m$. By Lemma B, at most $B_{j'} \le 2^{2^{t-1}-1}$ of the sets V_z are nonempty; let $U_t = V_z$ be the largest. To write r_i in $M[z]$, define $x_t = x_{t-1}$, $z'' \leftarrow i'$; to read r_i from $M[z]$, set $i' \leftarrow t$ and put $x_t = x_{z''}$ if z'' is defined, otherwise let x_t be the precomputed constant $M[z]$. In both cases $|U_t| \ge (|U_{t-1}| - 2^{t-1}m)/2^{2^{t-1}-1}$ is sufficiently large.

RAM
Paterson
2-adic chain
suffix parity function
breadth-first search
cache-friendly
sequential
linked
radix exchange sort

If $t < f$ we cannot be sure that $r_1 = \rho x$. The reason is that the set $W = \{x \in U_t \mid x \,\&\, \lfloor 2^{\lg n+s} - 2^s \rfloor \neq 0$ for some $s \in S_{1'}\}$ has size $|W| \le |S_{1'}| \lg n \le 2^{t+e+f}$, and $|U_t \setminus W| \ge 2^{2^{e+f}-2^t+1} - 2^{t+e+f} > 2^{2^t-1} \ge |\{x_{1'} \,\&\, \lfloor 2^{\lg n} - 1 \rfloor \mid x_0 \in U_t \setminus W\}|$. Two elements of $U_t \setminus W$ cannot have the same value of $\rho x = x_{1'} \,\&\, \lfloor 2^{\lg n} - 1 \rfloor$.

[The same lower bound applies even if we allow the RAM to make arbitrary $2^{2^{t-1}}$-way branches based on the contents of $(r_1, \ldots, r_l)$ at time t.]

125. Start as in answer 124, but with $U_0 = [0\,.\,.\,2^g)$. Simplifying that argument by eliminating the sets W will yield sets such that $|U_t| \ge 2^g/\max(2^m, 2n)^t$; for example, at most $2n$ different shift instructions can occur.

Suppose we can stop at time t with $t < \lfloor \lg(h+1) \rfloor$. The proof of Theorem P yields p and q with $x^R \,\&\, \lfloor 2^p - 2^q \rfloor$ independent of $x \,\&\, \lfloor 2^{p+s} - 2^{q+s} \rfloor$. Hence the hinted extension of Lemma B shows that x^R takes on at most $2^{2^t-1} \le 2^{(h-1)/2}$ different values, for every setting of the other bits $\{x \,\&\, \lfloor 2^{p+s} - 2^{q+s} \rfloor \mid s \in S_t\}$. Consequently $r_1 = x_{1'}$ can be the correct value of x^R for at most $2^{(h-1)/2+g-h}$ values of x. But $2^{(h-1)/2+g-h}$ is less than $|U_t|$, by (106).

126. M. S. Paterson has proposed a related (but different) conjecture: For every 2-adic chain with k addition-subtraction operations, there is a (possibly huge) integer x with $\nu x = k+1$ such that the chain does not calculate $2^{\lambda x}$.

127. Use exercise 110 to compute $[(1 \ll (\nu x)) \,\&\, \alpha \neq 0]$ for a suitable constant α. (The special case $\nu x = n$ may also need to be handled separately.)

128. The weaker lower bound $\Omega(\log\log n)$ follows from Theorem R, because $\rho x = \nu(x'-1)$ when $x' = x \,\&\, \overline{x-1}$.

129. Note that the suffix parity function $x^{\oplus}$ is considered in exercises 36 and 117.

130. If the answer is "no," the analogous question with *variable a* suggests itself.

131. This program does a typical "breadth-first search," keeping LINK(q) $=$ r. Register u is the vertex currently being examined; v is one of its successors.

0H LDOU r,q,link	1	r ← LINK(q).		STOU v,q,link	$\lvert R\rvert-\lvert Q\rvert$	LINK(q) ← v.
SET u,r	1	u ← r.		STOU r,v,link	$\lvert R\rvert-\lvert Q\rvert$	LINK(v) ← r.
1H LDOU a,u,arcs	$\lvert R\rvert$	a ← ARCS(u).		SET q,v	$\lvert R\rvert-\lvert Q\rvert$	q ← v.
BZ a,4F	$\lvert R\rvert$	Is $S[u] = \emptyset$?	3H	PBNZ a,2B	S	Loop on a.
2H LDOU v,a,tip	S	v ← TIP(a).	4H	LDOU u,u,link	$\lvert R\rvert$	u ← LINK(u).
LDOU a,a,next	S	a ← NEXT(a).		CMPU t,u,r	$\lvert R\rvert$	Is u ≠ r?
LDOU t,v,link	S	t ← LINK(v).		PBNZ t,1B	$\lvert R\rvert$	Continue if so.
PBNZ t,3F	S	Is v ∈ R?				∎

132. (a) We always have $\tau(U) \subseteq \&_{u \notin U}\, \delta_u = \sigma(U)$. And equality holds if and only if $2^u \subseteq \rho(u')$ for all $u \in U$ and $u' \in U$.

(b) We've proved that $\tau(U) \subseteq \sigma(U)$; hence $T \subseteq U$. And if $t \in T$ we have $2^t \subseteq \rho_u$ for all $u \in U$. Therefore $\sigma(T) \subseteq \tau(T)$.

(c) Parts (a) and (b) prove that the elements of C_n represent the cliques.

(d) If $u \subseteq v$ then $u \,\&\, \rho_k \subseteq v \,\&\, \rho_k$ and $u \,\&\, \delta_k \subseteq v \,\&\, \delta_k$; so we can work entirely with maximal entries. The following algorithm uses cache-friendly sequential (rather than linked) allocation, in a manner analogous to radix exchange sort (Algorithm 5.2.2R).

We assume that $w_1 \ldots w_s$ is a workspace of s unsigned words, bounded by $w_0 = 0$ and $w_{s+1} = 2^n - 1$. The elements of C^+_{k-1} appear initially in positions $w_1 \ldots w_m$, and our goal is to replace them by the elements of C^+_k.

M1. [Initialize.] Terminate if $\rho_k = 2^n - 1$. Otherwise set $v \leftarrow 2^k$, $i \leftarrow 1$, $j \leftarrow m$.

M2. [Partition on v.] While $w_i \,\&\, v = 0$, set $i \leftarrow i+1$. While $w_j \,\&\, v \neq 0$, set $j \leftarrow j-1$. Then if $i > j$, go to M3; otherwise swap $w_i \leftrightarrow w_j$, set $i \leftarrow i+1$, $j \leftarrow j-1$, and repeat this step.

M3. [Split $w_i \ldots w_m$.] Set $l \leftarrow j$, $p \leftarrow s+1$. While $i \leq m$, do subroutine Q with $u = w_i$ and set $i \leftarrow i+1$.

M4. [Combine maximal elements.] Set $m \leftarrow l$. While $p \leq s$, set $m \leftarrow m+1$, $w_m \leftarrow w_p$, and $p \leftarrow p+1$. ▮

Subroutine Q uses global variables j, k, l, p, and v. It essentially replaces the word u by $u' = u \,\&\, \rho_k$ and $u'' = u \,\&\, \delta_k$, retaining them if they are still maximal. If so, u' goes into the upper workspace $w_p \ldots w_s$ but u'' stays below.

Q1. [Examine u'.] Set $w \leftarrow u \,\&\, \rho_k$ and $q \leftarrow s$. If $w = u$, go to Q4.

Q2. [Is it comparable?] If $q < p$, go to Q3. Otherwise if $w \,\&\, w_q = w$, go to Q7. Otherwise if $w \,\&\, w_q = w_q$, go to Q4. Otherwise set $q \leftarrow q-1$ and repeat Q2.

Q3. [Tentatively accept u'.] Set $p \leftarrow p-1$ and $w_p \leftarrow w$. Memory overflow occurs if $p \leq m+1$. Otherwise go to Q7.

Q4. [Prepare for loop.] Set $r \leftarrow p$ and $w_{p-1} \leftarrow 0$.

Q5. [Remove nonmaximals.] While $w \mid w_q \neq w$, set $q \leftarrow q-1$. While $w \mid w_r = w$, set $r \leftarrow r+1$. Then if $q < r$, go to Q6; otherwise set $w_q \leftarrow w_r$, $w_r \leftarrow 0$, $q \leftarrow q-1$, $r \leftarrow r-1$, and repeat this step.

Q6. [Reset p.] Set $w_q \leftarrow w$ and $p \leftarrow q$. Terminate the subroutine if $w = u$.

Q7. [Examine u''.] Set $w \leftarrow u \,\&\, \bar{v}$. If $w = w_q$ for some q in the range $1 \leq q \leq j$, do nothing. Otherwise set $l \leftarrow l+1$ and $w_l \leftarrow w$. ▮

In practice this algorithm performs quite well; for example, when it is applied to the 8×8 queen graph (exercise 7–129), it finds the 310 maximal cliques after only 57283 mems of computation, using 397 words of workspace. It finds the 10188 maximal independent sets of that same graph after about 26 megamems, using 15090 words; there are respectively $(728, 6912, 2456, 92)$ such sets of sizes $(5, 6, 7, 8)$, including the 92 famous solutions to the eight queens problem.

Reference: N. Jardine and R. Sibson, *Mathematical Taxonomy* (Wiley, 1971), Appendix 5. Many other algorithms for listing maximal cliques have also been published. See, for example, W. Knödel, *Computing* **3** (1968), 239–240, **4** (1969), 75; C. Bron and J. Kerbosch, *CACM* **16** (1973), 575–577; S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, *SICOMP* **6** (1977), 505–517; E. Loukakis, *Computers and Math. with Appl.* **9** (1983), 583–589; D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *Inf. Proc. Letters* **27** (1988), 119–123. See also exercise 5–23.

133. (a) An independent set is a clique of $\overline{G}$; so complement G. (b) A vertex cover is the complement of an independent set; so complement G, then complement the outputs.

134. $a \mapsto 00$, $b \mapsto 01$, $c \mapsto 11$ is the first mapping of class II.

135. The unary operators are simple: $\neg(x_l x_r) = \bar{x}_r \bar{x}_l$; $\diamond(x_l x_r) = x_r x_r$; $\square(x_l x_r) = x_l x_l$. And $x_l x_r \Leftrightarrow y_l y_r = (z_l \vee z_r)(z_l \wedge z_r)$, where $z_l = x_l \oplus y_l$ and $z_r = x_r \oplus y_r$.

136. (a) Classes II, III, IV$_a$, and IV$_c$ all have the optimum cost 4. Curiously the functions $z_l = x_l \vee y_l \vee (x_r \wedge y_r)$, $z_r = x_r \vee y_r$ work for the mapping $(a,b,c) \mapsto (00, 01, 11)$ of class II as well as for the mapping $(a,b,c) \mapsto (00, 01, 1{*})$ of class IV$_c$. [This operation is equivalent to saturated addition, when $a = 0$, $b = 1$, and c stands for “more than 1.”]

(b) The symmetry between a, b, and c implies that we need only try classes I, IV$_a$, and V$_a$; and those classes turn out to cost 6, 7, and 8. One winner for class I, with

overflow
queen graph
maximal independent sets
eight queens problem
Jardine
Sibson
Knödel
Bron
Kerbosch
Tsukiyama
Ide
Ariyoshi
Shirakawa
Loukakis
Johnson
Yannakakis
Papadimitriou
complement
complement
saturated addition

$(a,b,c) \mapsto (00,01,10)$, is $z_l = v_r \wedge \bar{u}_l$, $z_r = v_l \wedge \bar{u}_r$, where $u_l = x_l \oplus y_l$, $u_r = x_r \oplus y_r$, $v_l = y_r \oplus u_l$, and $v_r = y_l \oplus u_r$. [See exercise 7.1.2–60, which gives the same answer but with $z_l \leftrightarrow z_r$. The reason is that we have $(x+y+z) \bmod 3 = 0$ in this problem but $(x+y-z) \bmod 3 = 0$ in that one; and $z_l \leftrightarrow z_r$ is equivalent to negation. The binary operation $z = x \circ y$ in this case can also be characterized by the fact that the elements (x,y,z) are all the same or all different; thus it is familiar to people who play the game of SET. It is the only binary operation on n-element sets that has $n!$ automorphisms and differs from the trivial examples $x \circ y = x$ or $x \circ y = y$.]

(c) Cost 3 is achieved only with class I: Let $(a,b,c) \mapsto (00,01,10)$ and $z_l = (x_l \vee x_r) \wedge y_l$, $z_r = \bar{x}_r \wedge y_r$.

137. In fact, $z = (x+1) \mathbin{\&} y$ when $(a,b,c) \mapsto (00,01,10)$. [It's a contrived example.]

138. The simplest case known to the author requires the calculation of *two* binary operations, such as

$$\begin{pmatrix} a & b & b \\ a & b & b \\ c & a & a \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a & b & a \\ a & b & a \\ c & a & c \end{pmatrix};$$

each has cost 2 in class V_a, but the costs are $(3,2)$ and $(2,3)$ in classes I and II.

139. The calculation of z_2 is essentially equivalent to exercise 136(b); so the natural representation (111) wins. Fortunately this representation also is good for z_1, with $z_{1l} = x_l \wedge y_l$, $z_{1r} = x_r \wedge y_r$.

140. With representation (111), first use full binary adders to compute $(a_1a_0)_2 = x_l + y_l + z_l$ and $(b_1b_0)_2 = x_r + y_r + z_r$ in $5+5=10$ steps. Now the "greedy footprint" method shows how to compute the four desired functions of (a_1,a_0,b_1,b_0) in eight further steps: $u_l = a_1 \wedge \bar{b}_0$, $u_r = a_0 \wedge \bar{b}_1$; $t_1 = a_1 \oplus b_0$, $t_2 = a_0 \oplus b_1$, $t_3 = a_1 \oplus t_2$, $t_4 = a_0 \oplus t_1$, $v_l = t_3 \wedge \bar{t}_1$, $v_r = t_4 \wedge \bar{t}_2$. [Is this method optimum?]

141. Suppose we've computed bits $a = a_0a_1 \ldots a_{2m-1}$ and $b = b_0b_1 \ldots b_{2m-1}$ such that

$$a_s = [s=1 \text{ or } s=2 \text{ or } s \text{ is a sum of distinct Ulam numbers} \le m \text{ in exactly one way}],$$
$$b_s = [s \text{ is a sum of distinct Ulam numbers} \le m \text{ in more than one way}],$$

for some integer $m = U_n \ge 2$. For example, when $m = n = 2$ we have $a = 0111$ and $b = 0000$. Then $\{s \mid s \le m \text{ and } a_s = 1\} = \{U_1, \ldots, U_n\}$; and $U_{n+1} = \min\{s \mid s > m \text{ and } a_s = 1\}$. (Notice that $a_s = 1$ when $s = U_{n-1} + U_n$.) The following simple bitwise operations preserve these conditions: $n \leftarrow n+1$, $m \leftarrow U_n$, and

$$(a_m \ldots a_{2m-1}, b_m \ldots b_{2m-1}) \leftarrow \big((a_m \ldots a_{2m-1} \oplus a_0 \ldots a_{m-1}) \mathbin{\&} \overline{b_m \ldots b_{2m-1}},$$
$$(a_m \ldots a_{2m-1} \mathbin{\&} a_0 \ldots a_{m-1}) \mid b_m \ldots b_{2m-1}\big),$$

where $a_s = b_s = 0$ for $2U_{n-1} \le s < 2U_n$ on the right side of this assignment.

[See M. C. Wunderlich, *BIT* **11** (1971), 217–224; *Computers in Number Theory* (1971), 249–257. These mysterious numbers, which were first defined by S. Ulam in *SIAM Review* **6** (1964), 348, have baffled number theorists for many years. The ratio U_n/n appears to converge to a constant, ≈ 13.52; for example, $U_{20000000} = 270371127$ and $U_{40000000} = 540752349$. Furthermore, D. W. Wilson has observed empirically that the numbers form quasi-periodic "clusters" whose centers differ by multiples of another constant, ≈ 21.6016. Calculations by Jud McCranie and the author for $U_n < 640000000$ indicate that the largest gap $U_n - U_{n-1}$ may occur between $U_{24576523} = 332250401$ and $U_{24576524} = 332251032$; the smallest gap $U_n - U_{n-1} = 1$ apparently occurs only when $U_n \in \{2,3,4,48\}$. Certain small gaps like 6, 11, 14, and 16 have never been observed.]

SET
greedy footprint
footprints
Wunderlich
Ulam
Wilson
McCranie
Knuth
gap

don't-cares
Breuer
Frey
MOR
triply linked tree
traversal in preorder

142. Algorithm E in that exercise performs the following operations on subcubes: (i) Count the $*$s in a given subcube c. (ii) Given c and c', test if $c \subseteq c'$. (iii) Given c and c', compute $c \sqcup c'$ (if it exists). Operation (i) is simple with sideways addition; let's see which of the nine classes of two-bit encodings (119), (123), (124) works best for (ii) and (iii). Suppose $a = 0$, $b = 1$, $c = *$; the symmetry between 0 and 1 means that we need only examine classes I, III, IV_a, IV_c, V_a, and V_c.

For the asterisks-and-bits mapping $(0,1,*) \mapsto (00,01,10)$, which belongs to class I, the truth table for $c \not\subseteq c'$ is $010{*}100{*}110{*}{*}{*}{*}{*}$ in each component. (For example, $0 \subseteq *$ and $* \not\subseteq 1$. The $*$s in this truth table are don't-cares for the unused codes 11.) The methods of Section 7.1.2 tell us that the cheapest such functions have cost 3; for example, $c \subseteq c'$ if and only if $((b \oplus b') \mid a) \mathbin{\&} \bar{a}' = 0$. Furthermore the consensus $c \sqcup c' = c''$ exists if and only if $\nu z = 1$, where $z = (b \oplus b') \mathbin{\&} {\sim}(a \oplus a')$. And in that case, $a'' = (a \oplus b \oplus b') \mathbin{\&} {\sim}(a \oplus a')$, $b'' = (b \mid b') \mathbin{\&} \bar{z}$. [The asterisk and bit codes were used for this purpose by M. A. Breuer in *Proc. ACM Nat. Conf.* **23** (1968), 241–250.]

But class III works out better, with $(0,1,*) \mapsto (01,10,00)$. Then $c \subseteq c'$ if and only if $(\bar{c}_l \mathbin{\&} c'_l) \mid (\bar{c}_r \mathbin{\&} c'_r) = 0$; $c \sqcup c' = c''$ exists if and only if $\nu z = 1$ where $z = x \mathbin{\&} y$, $x = c_l \mid c'_l$, $y = c_r \mid c'_r$; and $c''_l = x \oplus z$, $c''_r = y \oplus z$. We save two operations for each consensus, with respect to class I, compensating for an extra step when counting asterisks.

Classes IV_a, V_a, and V_c turn out to be far inferior. Class IV_c has some merit, but class III is best.

143. $f(x) = ((x \mathbin{\&} m_1) \ll 17) \mid ((x \gg 17) \mathbin{\&} m_1) \mid ((x \mathbin{\&} m_2) \ll 15) \mid ((x \gg 15) \mathbin{\&} m_2) \mid ((x \mathbin{\&} m_3) \ll 10) \mid ((x \gg 10) \mathbin{\&} m_3) \mid ((x \mathbin{\&} m_4) \ll 6) \mid ((x \gg 6) \mathbin{\&} m_4)$, where $m_1 = {}^{\#}\texttt{7f7f7f7f7f7f}$, $m_2 = {}^{\#}\texttt{fefefefefefe}$, $m_3 = {}^{\#}\texttt{3f3f3f3f3f3f3f}$, $m_4 = {}^{\#}\texttt{fcfcfcfcfcfcfc}$. [See, for example, *Chess Skill in Man and Machine*, edited by Peter W. Frey (1977), page 59. Five steps suffice to compute $f(x)$ on MMIX (four MOR operations and one OR), since $f(x) = q \cdot x \cdot q' \mid q' \cdot x \cdot q$ with $q = {}^{\#}\texttt{40a05028140a0502}$ and $q' = {}^{\#}\texttt{2010884422110804}$.]

144. Node $j \oplus (k \ll 1)$, where $k = j \mathbin{\&} -j$.

145. It names the ancestor of the leaf node $j \mid 1$ at height h.

146. By (136) we want to show that $\lambda(j \mathbin{\&} -i) = \rho l$ when $l - 2^{\rho l} < i \le l \le j < l + 2^{\rho l}$. The desired result follows from (35) because $-l \le -i < -l + 2^{\rho l}$.

147. (a) $\pi v_j = \beta v_j = j$, $\alpha v_j = 1 \ll \rho j$, and $\tau j = \Lambda$, for $1 \le j \le n$.

(b) Suppose $n = 2^{e_1} + \cdots + 2^{e_t}$ where $e_1 > \cdots > e_t \ge 0$, and let $n_k = 2^{e_1} + \cdots + 2^{e_k}$ for $0 \le k \le t$. Then $\pi v_j = j$ and $\beta v_j = \alpha v_j = n_k$ for $n_{k-1} < j \le n_k$. Also $\tau n_k = v_{n_{k-1}}$ for $1 \le k \le t$, where $v_0 = \Lambda$; all other $\tau j = \Lambda$.

148. Yes, if $\pi y_1 = 010000$, $\pi y_2 = 010100$, $\pi x_1 = 010101$, $\pi x_2 = 010110$, $\pi x_3 = 010111$, $\beta x_3 = 010111$, $\beta y_2 = 010100$, $\beta x_2 = 011000$, $\beta y_1 = 010000$, and $\beta x_1 = 100000$.

149. We assume that $\mathtt{CHILD}(v) = \mathtt{SIB}(v) = \mathtt{PARENT}(v) = \Lambda$ initially for all vertices v (including $v = \Lambda$), and that there is at least one nonnull vertex.

S1. [Make triply linked tree.] For each of the n arcs $u \to v$ (perhaps $v = \Lambda$), set $\mathtt{SIB}(u) \leftarrow \mathtt{CHILD}(v)$, $\mathtt{CHILD}(v) \leftarrow u$, $\mathtt{PARENT}(u) \leftarrow v$. (See exercise 2.3.3–6.)

S2. [Begin first traversal.] Set $p \leftarrow \mathtt{CHILD}(\Lambda)$, $n \leftarrow 0$, and $\lambda 0 \leftarrow -1$.

S3. [Compute β in the easy case.] Set $n \leftarrow n + 1$, $\pi p \leftarrow n$, $\tau n \leftarrow \Lambda$, and $\lambda n \leftarrow 1 + \lambda(n \gg 1)$. If $\mathtt{CHILD}(p) \ne \Lambda$, set $p \leftarrow \mathtt{CHILD}(p)$ and repeat this step; otherwise set $\beta p \leftarrow n$.

S4. [Compute τ, bottom-up.] Set $\tau\beta p \leftarrow \mathtt{PARENT}(p)$. Then if $\mathtt{SIB}(p) \ne \Lambda$, set $p \leftarrow \mathtt{SIB}(p)$ and return to S3; otherwise set $p \leftarrow \mathtt{PARENT}(p)$.

S5. [Compute β in the hard case.] If $p \neq \Lambda$, set $h \leftarrow \lambda(n \,\&\, -\pi p)$, then $\beta p \leftarrow ((n \gg h) \mid 1) \ll h$, and go back to S4.

S6. [Begin second traversal.] Set $p \leftarrow$ `CHILD`(Λ), $\lambda 0 \leftarrow \lambda n$, $\beta\Lambda \leftarrow \alpha\Lambda \leftarrow 0$.

S7. [Compute α, top-down.] Set $\alpha p \leftarrow \alpha($`PARENT`$(p)) \mid (\beta p \,\&\, -\beta p)$. Then if `CHILD`$(p) \neq \Lambda$, set $p \leftarrow$ `CHILD`(p) and repeat this step.

S8. [Continue to traverse.] If `SIB`$(p) \neq \Lambda$, set $p \leftarrow$ `SIB`(p) and go to S7. Otherwise set $p \leftarrow$ `PARENT`(p), and repeat step S8 if $p \neq \Lambda$. ▌

150. We may assume that the elements A_j are distinct, by regarding them as ordered pairs (A_j, j). The hinted binary search tree, which is a special case of the "Cartesian trees" introduced by Jean Vuillemin [*CACM* **23** (1980), 229–239], has the property that $k(i,j)$ is the nearest common ancestor of i and j. Indeed, the ancestors of any given node j are precisely the nodes k such that A_k is a right-to-left minimum of $A_1 \ldots A_j$ or A_k is a left-to-right minimum of $A_j \ldots A_n$.

The algorithm of the preceding answer does the desired preprocessing, except that we need to set up a triply linked tree differently on the nodes $\{0, 1, \ldots, n\}$. Start as before with `CHILD`(v) = `SIB`(v) = `PARENT`$(v) = 0$ for $0 \le v \le n$, and let $\Lambda = 0$. Assume that $A_0 \le A_j$ for $1 \le j \le n$. Set $t \leftarrow 0$ and do the following steps for $v = n$, $n-1, \ldots, 1$: Set $u \leftarrow 0$; then while $A_v < A_t$ set $u \leftarrow t$ and $t \leftarrow$ `PARENT`(t). If $u \neq 0$, set `SIB`$(v) \leftarrow$ `SIB`(u), `SIB`$(u) \leftarrow 0$, `PARENT`$(u) \leftarrow v$, `CHILD`$(v) \leftarrow u$; otherwise simply set `SIB`$(v) \leftarrow$ `CHILD`(t). Also set `CHILD`$(t) \leftarrow v$, `PARENT`$(v) \leftarrow t$, $t \leftarrow v$.

Continue with step S2 after the tree has been built. The running time is $O(n)$, because the operation $t \leftarrow$ `PARENT`(t) is performed at most once for each node t. [This beautiful way to reduce the range minimum query problem to the nearest common ancestor problem was discovered by H. N. Gabow, J. L. Bentley, and R. E. Tarjan, *STOC* **16** (1984), 137–138, who also suggested the following exercise.]

151. For node v with k children $u_1, \ldots, u_k$, define the node sequence $S(v) = v$ if $k = 0$; $S(v) = vS(u_1)$ if $k = 1$; and $S(v) = S(u_1)v \ldots vS(u_k)$ if $k > 1$. (Consequently v appears exactly $\max(k-1, 1)$ times in $S(v)$.) If there are k trees in the forest, rooted at $u_1, \ldots, u_k$, write down the node sequence $S(u_1)\Lambda \ldots \Lambda S(u_k) = V_1 \ldots V_N$. (The length of this sequence will satisfy $n \le N < 2n$.) Let A_j be the depth of node V_j, for $1 \le j \le N$, where Λ has depth 0. (For example, consider the forest (141), but add another child $K \to D$ and an isolated node L. Then $V_1 \ldots V_{15} = CFAGJDHDK\Lambda BEI\Lambda L$ and $A_1 \ldots A_{15} = 231342323012301$.) The nearest common ancestor of u and v, when $u = V_i$ and $v = V_j$, is then $V_{k(i,j)}$ in the range minimum query problem. [See J. Fischer and V. Heun, *Lecture Notes in Comp. Sci.* **4009** (2006), 36–48.]

152. Step V1 finds the level above which αx and αy have bits that apply to both of their ancestors. (See exercise 148.) Step V2 increases h, if necessary, to the level where they have a common ancestor, or to the top level λn if they don't (namely if $k = 0$). If $\beta x \neq \beta z$, step V4 finds the topmost level among x's ancestors that leads to level h; hence it knows the lowest ancestor $\hat{x}$ for which $\beta\hat{x} = \beta z$ (or $\hat{x} = \Lambda$). Finally in V5, preorder tells us which of $\hat{x}$ or $\hat{y}$ is an ancestor of the other.

153. That pointer has ρj bits, so it ends after $\rho 1 + \rho 2 + \cdots + \rho j = j - \nu j$ bits of the packed string, by (61). [Here j is even. Navigation piles were introduced in *Nordic Journal of Computing* **10** (2003), 238–262.]

154. The gray lines define 36°-36°-90° triangles, ten of which make a pentagon with 72° angles at each vertex. These pentagons tile the hyperbolic plane in such a way that *five* of them meet at each vertex.

traversal in postorder
Cartesian trees
Vuillemin
right-to-left minimum
left-to-right minimum
triply linked tree
Gabow
Bentley
Tarjan
Fischer
Heun
sum of rho

155. Observe first that $0 \le (\alpha 0)_{1/\phi} < \phi^{-1} + \phi^{-3} + \phi^{-5} + \cdots = 1$, since there are no consecutive 1s. Observe next that $F_{-n}\phi \equiv \phi^{-n}$ (modulo 1), by exercise 1.2.8–11. Now add $F_{k_1}\phi + \cdots + F_{k_r}\phi$. For example, $(4\phi) \bmod 1 = \phi^{-5} + \phi^{-2}$; $(-2\phi) \bmod 1 = \phi^{-4} + \phi^{-1}$.

This argument also proves the interesting formula $\lfloor N(\alpha)\phi \rfloor = -N(\alpha 0)$.

156. (a) Start with $y \leftarrow 0$, and with k large enough that $|x| < F_{k+1}$. If $x < 0$, set $k \leftarrow (k-1) \mid 1$, and while $x + F_k > 0$ set $k \leftarrow k - 2$; then set $y \leftarrow y + (1 \ll k)$, $x \leftarrow x + F_{k+1}$; repeat. Otherwise if $x > 1$, set $k \leftarrow k \mathbin{\&} -2$, and while $x - F_k \le 0$ set $k \leftarrow k - 2$; then set $y \leftarrow y + (1 \ll k)$, $x \leftarrow x - F_{k+1}$; repeat. Otherwise set $y \leftarrow y + x$ and terminate with $y = (\alpha)_2$.

(b) The operations $x_1 \leftarrow a_1$, $y_1 \leftarrow -a_1$, $x_k \leftarrow y_{k-1} + a_k$, $y_k \leftarrow x_{k-1} - x_k$ compute $x_k = N(a_1 \ldots a_k)$ and $y_k = N(a_1 \ldots a_k 0)$. [Does *every* broadword chain for $N(a_1 \ldots a_n)$ require $\Omega(n)$ steps?]

157. The laws are obvious except for the two cases involving $(\alpha-)$. For those we have $N((\alpha-)0^k) = N(\alpha 0^k) + F_{-k-2}$ for all $k \ge 0$, because decrementation never "borrows" at the right. (But the analogous formula $N((\alpha+)0^k) = N(\alpha 0^k) + F_{-k-1}$ does *not* hold.)

158. Incrementation satisfies the rules $(\alpha 00)+ = \alpha 01$, $(\alpha 10)+ = (\alpha+)00$, $(\alpha 1)+ = (\alpha+)0$. It can be achieved with six 2-adic operations on the integer $x = (\alpha)_2$ by setting $y \leftarrow x \mid (x \gg 1)$, $z \leftarrow y \mathbin{\&} {\sim}(y+1)$, $x \leftarrow (x \mid z) + 1$.

Decrementation of a nonzero codeword is more difficult. It satisfies $(\alpha 10^{2k})- = \alpha 0(10)^k$, $(\alpha 10^{2k+1})- = \alpha(01)^{k+1}$; hence by Corollary I it cannot be computed by a 2-adic chain. Yet six operations suffice, if we allow monus: $y \leftarrow x - 1$, $z \leftarrow y \mathbin{\&} \bar{x}$, $w \leftarrow z \mathbin{\&} \mu_0$, $x \leftarrow y - w + (w \mathbin{\dot-} (z - w))$.

159. Besides the Fibonacci number system (146) and the negaFibonacci number system (147), there's also an *odd Fibonacci number system*: *Every positive integer x can be written uniquely in the form*

$$x = F_{l_1} + F_{l_2} + \cdots + F_{l_s}, \qquad \text{where } l_1 \ggcurly l_2 \ggcurly \cdots \ggcurly l_s > 0 \text{ and } l_s \text{ is odd.}$$

Given a negaFibonacci code α, the following 20-step 2-adic chain converts $x = (\alpha)_2$ to $y = (\beta)_2$ to $z = (\gamma)_2$, where β is the odd codeword with $N(\alpha) = F(\beta)$ and γ is the standard codeword with $F(\beta) = F(\gamma 0)$: $x^+ \leftarrow x \mathbin{\&} \mu_0$, $x^- \leftarrow x \oplus x^+$; $d \leftarrow x^+ - x^-$; $t \leftarrow d \mid x^-$, $t \leftarrow t \mathbin{\&} {\sim}(t \ll 1)$; $y \leftarrow (d \mathbin{\&} \bar{\mu}_0) \oplus t \oplus ((t \mathbin{\&} x^-) \gg 1)$; $z \leftarrow (y+1) \gg 1$; $w \leftarrow z \oplus (4\mu_0)$; $t \leftarrow w \mathbin{\&} {\sim}(w+1)$; $z \leftarrow z \oplus (t \mathbin{\&} (z \oplus ((w+1) \gg 1)))$.

Corresponding negaFibonacci and odd representations satisfy the remarkable law

$$F_{k_1+m} + \cdots + F_{k_r+m} = (-1)^m (F_{l_1-m} + \cdots + F_{l_s-m}), \qquad \text{for all integers } m.$$

For example, if $N(\alpha) < 0$ the steps above will convert $x = (\alpha 0)_2$ to $y = (\beta)_2$, where $F((\beta \gg 2)0) = -N(\alpha)$. Furthermore β is the odd code for negaFibonacci α if and only if α^R is the odd code for negaFibonacci β^R, when $|\alpha| = |\beta|$ is odd and $N(\alpha) > 0$.

No finite 2-adic chain will go the other way, by Corollary I, because the Fibonacci code 10^k corresponds to negaFibonacci 10^{k+1} when k is odd, $(10)^{k/2}1$ when k is even. But if γ is a standard Fibonacci codeword we can compute $y = (\beta)_2$ from $z = (\gamma)_2$ by setting $y \leftarrow z \ll 1$, $t \leftarrow y \mathbin{\&} (y-1) \mathbin{\&} \bar{\mu}_0$, $y \leftarrow y - t + [t \ne 0]((t-1) \mathbin{\&} \mu_0)$. And then the method above will compute α^R from β^R. The overall running time for conversion to negaFibonacci form will then be of order $\log|\gamma|$, for two string reversals.

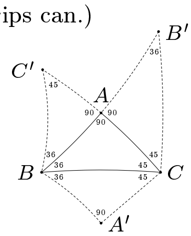
160. The text's rules are actually incomplete: They should also define the orientation of each neighbor. Let us stipulate that $\alpha_{sn} = \alpha$; $\alpha_{en} = \alpha$; $(\alpha 0)_{wn} = \alpha 0$, $(\alpha 1)_{wo} = \alpha 1$; $(\alpha 00)_{ns} = \alpha 00$, $(\alpha 10)_{nw} = \alpha 10$, $(\alpha 1)_{ne} = \alpha 1$; $(\alpha 0)_{oo} = \alpha 0$, $(\alpha 101)_{oo} = \alpha 101$,

broadword chain
2-adic chain
monus
magic mask
odd Fibonacci number system
2-adic chain
magic mask
string reversals

bipartite
cylinder
hyperbolic plane
upper halfplane
reflection
breadth-first search
Schläfli

$(\alpha 1001)_{oo} = \alpha 1001$, $(\alpha 0001)_{ow} = \alpha 0001$. Then a case analysis proves that all cells within d steps of the starting cell have a consistent labeling and orientation, by induction on the graph distance d. (Note the identity $\alpha{+} = ((\alpha 0){-}) \gg 1$.) Furthermore the labeling remains consistent when we attach y coordinates and move when necessary from one strip to another via the δ-rules of (153).

161. Yes, it is bipartite, because all of its edges are defined by the set of boundary lines. (The hyperbolic *cylinder* cannot be bicolored; but two adjacent strips can.)



162. It's convenient to view the hyperbolic plane through another lens, by mapping its points to the upper halfplane $\Im z > 0$. Then the "straight lines" become semicircles centered on the x-axis, together with vertical halflines as a limiting case. In this representation, the edges $|z-1| = \sqrt{2}$, $|z| = r$, and $\Re z = 0$ define a 36°-45°-90° triangle if $r^2 = \phi + \sqrt{\phi}$. Every triangle ABC has three neighbors CBA', ACB', and BAC', obtained by "reflecting" two of its edges about the third, where the reflection of $|z-c'| = r'$ about $|z-c| = r$ is $|z - c - \frac{1}{2}(x_1+x_2)| = \frac{1}{2}|x_1 - x_2|$, $x_j = r^2/(c' \pm r' - c)$.

The mapping $z \mapsto (z - z_0)/(z - \bar{z}_0)$ takes the upper halfplane into the unit circle; when $z_0 = \frac{1}{2}(\sqrt{\phi} - 1/\phi)(1 + 5^{1/4} i)$ the central pentagon will be symmetric. Repeated reflections of the initial triangle, using breadth-first search until reaching triangles that are invisible, will lead to Fig. 14. To get just the pentagons (without the gray lines), one can begin with just the central cell and perform reflections about *its* edges, etc.

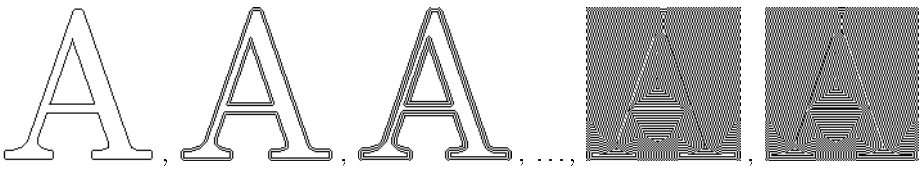
163. (This figure can be drawn as in exercise 162, starting with vertices that project to the three points ir, $ir\omega$, and $ir\omega^2$, where $r^2 = \frac{1}{2}(1+\sqrt{2})(4 - \sqrt{2} - \sqrt{6})$ and $\omega = e^{2\pi i/3}$. Using a notation devised by L. Schläfli in 1852, it can be described as the infinite tiling with parameters $\{3,8\}$, meaning that eight triangles meet at every vertex; see Schläfli's *Gesammelte Mathematische Abhandlungen* **1** (1950), 212. Similarly, the pentagrid and the tiling of exercise 154 have Schläfli symbols $\{5,4\}$ and $\{5,5\}$, respectively.)

164. The original definition requires more computation, even though it can be factored:

$$\text{custer}'(X) = X \mathbin{\&} {\sim}(Y_{\rm N} \mathbin{\&} Y \mathbin{\&} Y_{\rm S}), \qquad Y = X_{\rm W} \mathbin{\&} X \mathbin{\&} X_{\rm E}.$$

But the main reason for preferring (157) is that it produces a thinner, kingwise connected border. The rookwise connected border that results from the 1957 definition is less attractive, because it's noticeably darker when the border travels diagonally than when it travels horizontally or vertically. (Try some experiments and you'll see.)

165. The first image $X^{(1)}$ is the "outer" border of the original black pixels. Fingerprint-like whorls are formed thereafter. For example, starting with Fig. 15(a) we get



in a 120×120 bitmap, eventually alternating endlessly between two bizarre patterns. (Does *every* nonempty $M \times N$ bitmap lead to such a 2-cycle?)

166. If $X = \text{custer}(X)$, the sum of the elements of $X + (X \mathbin{⩓} 1) + (X \ll 1) + (X \gg 1) + (X \mathbin{⩔} 1)$ is at most $4MN + 2M + 2N$, since it is at most 4 in each cell of the rectangle and at most 1 in the adjacent cells. This sum is also five times the number of black pixels. Hence $f(M,N) \le \frac{4}{5}MN + \frac{2}{5}M + \frac{2}{5}N$. Conversely we get $f(M,N) \ge \frac{4}{5}MN - \frac{2}{5}$ by

letting the pixel in row i and column j be black unless $(i+2j) \bmod 5 = 2$. (This problem is equivalent to finding a minimum dominating set of the $M \times N$ grid.)

167. (a) With 17 steps we can construct a half adder and three full adders (see 7.1.2–(23)) so that $(z_1z_2)_2 = x_{\rm NW} + x_{\rm W} + x_{\rm SW}$, $(z_3z_4)_2 = x_{\rm N} + x_{\rm S}$, $(z_5z_6)_2 = x_{\rm NE} + x_{\rm E} + x_{\rm SE}$, and $(z_7z_8)_2 = z_2 + z_4 + z_6$. Then $f = S_1(z_1, z_3, z_5, z_7) \wedge (x \vee z_8)$, where the symmetric function f_1 needs seven operations by Fig. 9 in Section 7.1.2. [This solution is based on ideas of W. F. Mann and D. Sleator.]

(b) Given $x^- = X_{j-1}^{(t)}$, $x = X_j^{(t)}$, and $x^+ = X_{j+1}^{(t)}$, compute $a \leftarrow x^- \mathbin{\&} x^+$ $(= z_3)$, $b \leftarrow x^- \oplus x^+$ $(= z_4)$, $c \leftarrow x \oplus b$, $d \leftarrow c \gg 1$ $(= z_6)$, $c \leftarrow c \ll 1$ $(= z_2)$, $e \leftarrow c \oplus d$, $c \leftarrow c \mathbin{\&} d$, $f \leftarrow b \mathbin{\&} e$, $f \leftarrow f \mid c$ $(= z_7)$, $e \leftarrow b \oplus e$ $(= z_8)$, $c \leftarrow x \mathbin{\&} b$, $c \leftarrow c \mid a$, $b \leftarrow c \ll 1$ $(= z_5)$, $c \leftarrow c \gg 1$ $(= z_1)$, $d \leftarrow b \mathbin{\&} c$, $c \leftarrow b \mid c$, $b \leftarrow a \mathbin{\&} f$, $f \leftarrow a \mid f$, $f \leftarrow d \mid f$, $c \leftarrow b \mid c$, $f \leftarrow f \oplus c$ $(= S_1(z_1, z_3, z_5, z_7))$, $e \leftarrow e \mid x$, $f \leftarrow f \mathbin{\&} e$.

[For excellent summaries of the joys and passions of Life, including a proof that any Turing machine can be simulated, see Martin Gardner, *Wheels, Life and Other Mathematical Amusements* (1983), Chapters 20–22; E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways* **4** (A. K. Peters, 2004), Chapter 25.]

At last I've got what I wanted — an apparently unpredictable law of genetics.
. . . Overpopulation, like underpopulation, tends to kill.
A healthy society is neither too dense nor too sparse.

— JOHN H. CONWAY, letter to Martin Gardner (March 1970)

168. The following algorithm, which uses four n-bit registers x^-, x, x^+, and y, works properly even when $M = 1$ or $N = 1$. It needs only about two reads and two writes per raster word to transform $X^{(t)}$ to $X^{(t+1)}$ in (158):

C1. [Loop on k.] Do step C2 for $k = 1, 2, \ldots, N'$; then go to C5.

C2. [Loop on j.] Set $x \leftarrow A_{(M-1)k}$, $x^+ \leftarrow A_{0k}$, and $A_{Mk} \leftarrow x^+$. Then perform steps C3 and C4 for $j = 0, 1, \ldots, M-1$.

C3. [Move down.] Set $x^- \leftarrow x$, $x \leftarrow x^+$, and $x^+ \leftarrow A_{(j+1)k}$. (Now $x = A_{jk}$, and x^- holds the former value of $A_{(j-1)k}$.) Compute the bitwise function values $y \leftarrow f(x^- \gg 1, x^-, x^- \ll 1, x \gg 1, x, x \ll 1, x^+ \gg 1, x^+, x^+ \ll 1)$.

C4. [Update A_{jk}.] Set $x^- \leftarrow A_{j(k-1)} \mathbin{\&} -2$, $y \leftarrow y \mathbin{\&} (2^{n-1}-1)$, $A_{j(k-1)} \leftarrow x^- + (y \gg (n-2))$, $A_{jk} \leftarrow y + (x^- \ll (n-2))$.

C5. [Wrap around.] For $0 \le j < M$, set $x \leftarrow A_{jN'} \mathbin{\&} -2^{n-1-d}$, $A_{jN'} \leftarrow x + (A_{j1} \gg d)$, and $A_{j1} \leftarrow A_{j1} + (x \ll d)$, where $d = 1 + (N-1) \bmod (n-2)$. ▌

[An $M \times N$ torus is equivalent to an $(M-1) \times (N-1)$ array surrounded by zeros, in many cases like (157) and (159) and even (161). For exercise 173 we can clean an $(M-2) \times (N-2)$ array that is bordered by two rows and columns of zeros. But Life images (exercise 167) can grow without bound; they can't safely be confined to a torus.]

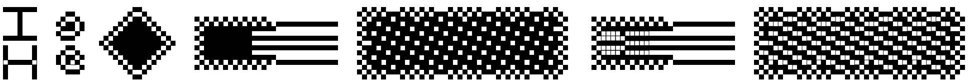
169. It quickly morphs into a rabbit, which proceeds to explode. Beginning at time 278, all activity stabilizes to a two-cycle formed from a set of traffic lights and three additional blinkers, together with three still lifs (tub, boat, and bee hive).

170. If $M \ge 2$ and $N \ge 2$, the first step blanks out the top row and the rightmost column. Then if $M \ge 3$ and $N \ge 3$, the next step blanks out the bottom row and the leftmost column. So in general we're left after $t = \min(M, N) - 1$ steps with a single row or column of black pixels: The first $\lceil t/2 \rceil$ rows, the last $\lceil t/2 \rceil$ columns, the last $\lfloor t/2 \rfloor$ rows, and the first $\lfloor t/2 \rfloor$ columns have been set to zero. The automaton will stop after making two more (nonproductive) cycles.

dominating set
grid
half adder
full adders
Mann
Sleator
Turing machine
Gardner
Berlekamp
Conway
Guy
CONWAY
Gardner
clean

171. Without (160): $x_1 \leftarrow x_{\rm SE} \,\&\, \bar{x}_{\rm N}$, $x_2 \leftarrow x_{\rm N} \,\&\, \bar{x}_{\rm SE}$, $x_3 \leftarrow x_{\rm E} \,\&\, \bar{x}_1$, $x_4 \leftarrow x_{\rm NE} \,\&\, \bar{x}_2$, $x_5 \leftarrow x_3 \mid x_4$, $x_6 \leftarrow x_{\rm W} \,\&\, \bar{x}_5$, $x_7 \leftarrow x_1 \,\&\, \bar{x}_{\rm NE}$, $x_8 \leftarrow x_7 \,\&\, \bar{x}_{\rm NW}$, $x_9 \leftarrow x_{\rm E} \mid x_{\rm SW}$, $x_{10} \leftarrow x_8 \,\&\, x_9$, $x_{11} \leftarrow x_{10} \mid x_6$, $x_{12} \leftarrow x_{\rm S} \,\&\, x_{11}$, $x_{13} \leftarrow x_2 \,\&\, \bar{x}_{\rm E}$, $x_{14} \leftarrow x_{13} \,\&\, x_{\rm W}$, $x_{15} \leftarrow x_{\rm N} \,\&\, x_{\rm NE}$, $x_{16} \leftarrow x_{\rm SW} \,\&\, x_{\rm W}$, $x_{17} \leftarrow x_{15} \mid x_{16}$, $x_{18} \leftarrow x_{\rm NE} \,\&\, x_{\rm SW}$, $x_{19} \leftarrow x_{17} \,\&\, \bar{x}_{18}$, $x_{20} \leftarrow x_{\rm E} \mid x_{\rm SE}$, $x_{21} \leftarrow x_{20} \mid x_{\rm S}$, $x_{22} \leftarrow x_{\rm NW} \,\&\, \bar{x}_{21}$, $x_{23} \leftarrow x_{22} \,\&\, x_{19}$, $x_{24} \leftarrow x_{12} \mid x_{14}$, $g \leftarrow x_{23} \mid x_{24}$. With (160), set $x_4 \leftarrow x_{\rm NE} \,\&\, \bar{x}_{\rm N}$ and leave everything else the same.

172. The statement isn't quite true; consider the following examples:



The 'I' and 'H' at the left show that pixels are sometimes left intact where paths join, and that rotating by 90° can make a difference. The next two examples illustrate a quirky influence of left-right reflection. The diamond example demonstrates that very thick images can be unthinnable; none of its black pixels can be removed without changing the number of holes. The final examples, one of which was inspired by the answer to exercise 166, were processed first without (160), in which case they are unchanged by the transformation. But with (160) they're thinned dramatically.

173. (a) If X and Y are closed, $X \,\&\, Y$ is closed; if X and Y are open, $X \mid Y$ is open. The hinted statement follows. Furthermore $X^{DD} = X^D$, because X^D is closed; similarly $X^{LL} = X^L$. (In fact we have $X^L = {\sim}({\sim}X)^D$, because the definitions are dual, obtained by swapping black with white.) Now $X^{DL} \subseteq X^D$, so $X^{DLD} \subseteq X^{DD} = X^D$. Dually, $X^L \subseteq X^{LDL}$. We conclude that there's no reason to launder a clean picture: $X^{DLDL} = (X^{DLD})^L \subseteq X^{DL} \subseteq (X^D)^{LDL} = X^{DLDL}$.

(b) We have $X^D = (X \mid X_{\rm W} \mid X_{\rm NW} \mid X_{\rm N}) \,\&\, (X \mid X_{\rm N} \mid X_{\rm NE} \mid X_{\rm E}) \,\&\, (X \mid X_{\rm E} \mid X_{\rm SE} \mid X_{\rm S}) \,\&\, (X \mid X_{\rm S} \mid X_{\rm SW} \mid X_{\rm W})$. Furthermore, in analogy with answer 167(b), this function can be computed from x^-, x, and x^+ in ten broadword steps: $f \leftarrow x \mid (x \gg 1) \mid ((x^- \mid (x^- \gg 1)) \,\&\, (x^+ \mid (x^+ \gg 1)))$, $f \leftarrow f \,\&\, (f \ll 1)$. [This answer incorporates ideas of D. R. Fuchs.]

To get X^L, just interchange | and &. For further discussion, see C. Van Wyk and D. E. Knuth, Report STAN-CS-79-707 (Stanford Univ., 1979), 15–36.]

174. Three-dimensional digital topology has been studied by R. Malgouyres, *Theoretical Computer Science* **186** (1997), 1–41.

175. There are 25 in the outline, $2+3$ in the eyes, $1+1$ in the ears, 4 in the nose, and 1 in the smile, totalling 37. (All white pixels are connected kingwise to the background.)

176. (a) If v isn't isolated, there are eight easy cases to consider, depending on what kind of neighbor v has in G.

(b) There's a vertex $w' \in G'$ adjacent to each vertex of $N_u \cup N_v$. (Four cases.)

(c) Yes. In fact, by definition (161), we always have $|S'(v')| \geq 2$.

(d) Let $N'_{v'} = \{v \mid v' \in N_v\}$. If v' is the east neighbor of u', call it $u'_{\rm E}$, either $u' \in G$ or $u'_{\rm S} \in G$; this element is adjacent to every vertex of $N'_{u'} \cup N'_{v'}$. A similar argument applies when $v' = u'_{\rm N}$. If $v' = u'_{\rm NE}$, there's no problem if $u' \in G$. Otherwise $u'_{\rm W} \in G$, $u'_{\rm S} \in G$, and either $u'_{\rm N} \in G$ or $u'_{\rm E} \in G$; hence $N'_{u'} \cup N'_{v'}$ is connected in G. Finally if $v' = u'_{\rm NE}$, the proof is easy if $u'_{\rm S} \in G$; otherwise $u' \in G$ and $v' \in G$.

(e) Given a nontrivial component C of G, with $v \in C$ and $v' \in S(v)$, let C' be the component of G' that contains v'. This component C' is well defined, by (a) and (b). Given a component C' of G', with $v' \in C'$ and $v \in S'(v')$, let C be the component of G that contains v. This component C is nontrivial and well defined, by (c) and (d). Finally, the correspondence $C \leftrightarrow C'$ is one-to-one.

dual
Fuchs
Van Wyk
Knuth
Malgouyres

177. Now the vertices of G are the *white* pixels, adjacent when they are *rook*-neighbors. So we define $N_{(i,j)} = \{(i,j),(i-1,j),(i,j+1)\}$. Arguments like those of answer 176, but simpler, establish a one-to-one correspondence between the nontrivial components of G and the components of G'.

178. Observe that in adjacent rows of X^*, two pixels of the same value are kingwise neighbors only if they are rookwise connected.

179. The pixels of each row $x_1 \ldots x_N$ can be "runlength encoded" as a sequence of integers $0 = c_0 < c_1 < \cdots < c_{2m+1} = N+2$ so that $x_j = 0$ for $j \in [c_0 \,.\,.\, c_1) \cup [c_2 \,.\,.\, c_3) \cup \cdots \cup [c_{2m} \,.\,.\, c_{2m+1})$ and $x_j = 1$ for $j \in [c_1 \,.\,.\, c_2) \cup \cdots \cup [c_{2m-1} \,.\,.\, c_{2m})$. (The number of runs per row tends to be reasonably small in most images. Notice that the background condition $x_0 = x_{N+1} = 0$ is implicitly assumed.)

The algorithm below uses a modified encoding with $a_j = 2c_j - (j \bmod 2)$ for $0 \le j \le 2m+1$. For example, the second row of the Cheshire cat has $(c_1, c_2, c_3, c_4, c_5) = (5, 8, 23, 25, 32)$; we will use $(a_1, a_2, a_3, a_4, a_5) = (9, 16, 45, 50, 63)$ instead. The reason is that white runs of adjacent rows are rookwise adjacent if and only if the corresponding intervals $[a_j \,.\,.\, a_{j+1})$ and $[b_k \,.\,.\, b_{k+1})$ overlap, and exactly the same condition characterizes when black runs of adjacent rows are kingwise adjacent. Thus the modified encoding nicely unifies both cases (see exercise 178).

We construct a triply linked tree of components, where each node has several fields: `CHILD`, `SIB`, and `PARENT` (tree links); `DORMANT` (a circular list of all children that aren't connected to the current row); `HEIR` (a node that has absorbed this one); `ROW` and `COL` (location of the first pixel); and `AREA` (the total number of pixels in the component).

The algorithm traverses the tree in *double order* (see exercise 2.3.1–18), using pairs of pointers $(\mathtt{P}, \mathtt{P}')$, where $\mathtt{P}' = \mathtt{P}$ when `P` is traversed the first time, $\mathtt{P}' = \mathtt{PARENT(P)}$ when `P` is traversed the second time. The successor of $(\mathtt{P}, \mathtt{P}')$ is $(\mathtt{Q}, \mathtt{Q}') = \mathrm{next}(\mathtt{P}, \mathtt{P}')$, determined as follows: If $\mathtt{P} = \mathtt{P}'$ and $\mathtt{CHILD(P)} \ne \Lambda$, then $\mathtt{Q} \leftarrow \mathtt{Q}' \leftarrow \mathtt{CHILD(P)}$; otherwise $\mathtt{Q} \leftarrow \mathtt{P}$ and $\mathtt{Q}' \leftarrow \mathtt{PARENT(Q)}$. If $\mathtt{P} \ne \mathtt{P}'$ and $\mathtt{SIB(P)} \ne \Lambda$, then $\mathtt{Q} \leftarrow \mathtt{Q}' \leftarrow \mathtt{SIB(P)}$; otherwise $\mathtt{Q} \leftarrow \mathtt{PARENT(P)}$ and $\mathtt{Q}' \leftarrow \mathtt{PARENT(Q)}$.

When there are m black runs, the tree will have $m+1$ nodes, not counting nodes that are dormant or have been absorbed. Moreover, the primed pointers $\mathtt{P}'_1, \ldots, \mathtt{P}'_{2m+1}$ of the double traversal $(\mathtt{P}_1, \mathtt{P}'_1), \ldots, (\mathtt{P}_{2m+1}, \mathtt{P}'_{2m+1})$ are precisely the components of the current row, in left-to-right order. For example, in (163) we have $m = 5$; and $(\mathtt{P}'_1, \ldots, \mathtt{P}'_{11})$ point respectively to ⓪, 🅑, ①, 🅑, ⓪, 🅒, ⓪, 🅐, ②, 🅐, ⓪.

I1. [Initialize.] Set $t \leftarrow 1$, $\mathtt{ROOT} \leftarrow \mathtt{LOC(NODE(0))}$, $\mathtt{CHILD(ROOT)} \leftarrow \mathtt{SIB(ROOT)} \leftarrow \mathtt{PARENT(ROOT)} \leftarrow \mathtt{DORMANT(ROOT)} \leftarrow \mathtt{HEIR(ROOT)} \leftarrow \Lambda$; also $\mathtt{ROW(ROOT)} \leftarrow \mathtt{COL(ROOT)} \leftarrow 0$, $\mathtt{AREA(ROOT)} \leftarrow N+2$, $s \leftarrow 0$, $a_0 \leftarrow b_0 \leftarrow 0$, $a_1 \leftarrow 2N+3$.

I2. [Input a new row.] Terminate if $s > M$. Otherwise set $b_k \leftarrow a_k$ for $k = 1, 2, \ldots$, until $b_k = 2N+3$; then set $b_{k+1} \leftarrow b_k$ as a "stopper." Set $s \leftarrow s+1$. If $s > M$, set $a_1 \leftarrow 2N+3$; otherwise let $a_1, \ldots, a_{2m+1}$ be the modified runlength encoding of row s as discussed above. (This encoding can be obtained with the help of the ρ function; see (43).) Set $j \leftarrow k \leftarrow 1$ and $\mathtt{P} \leftarrow \mathtt{P}' \leftarrow \mathtt{ROOT}$.

I3. [Gobble up short b's.] If $b_{k+1} \ge a_j$, go to I9. Otherwise set $(\mathtt{Q}, \mathtt{Q}') \leftarrow \mathrm{next}(\mathtt{P}, \mathtt{P}')$, $(\mathtt{R}, \mathtt{R}') \leftarrow \mathrm{next}(\mathtt{Q}, \mathtt{Q}')$, and do a four-way branch to (I4, I5, I6, I7) according as $2[\mathtt{Q} \ne \mathtt{Q}'] + [\mathtt{R} \ne \mathtt{R}'] = (0, 1, 2, 3)$.

I4. [Case 0.] (Now $\mathtt{Q} = \mathtt{Q}'$ is a child of $\mathtt{P}'$, and $\mathtt{R} = \mathtt{R}'$ is the first child of $\mathtt{Q}'$. Node `Q` will remain a child of $\mathtt{P}'$, but it will be preceded by any children of `R`.) Absorb `R` into $\mathtt{P}'$ (see below). Set $\mathtt{CHILD(Q)} \leftarrow \mathtt{SIB(R)}$ and $\mathtt{Q}' \leftarrow \mathtt{CHILD(R)}$. If $\mathtt{Q}' \ne \Lambda$,

runlength encoded
runlength encoding, see also edge transition
Cheshire cat
triply linked tree
circular list
double order
ruler function

set $\texttt{R} \leftarrow \texttt{Q}'$, and while $\texttt{R} \neq \Lambda$ set $\texttt{PARENT(R)} \leftarrow \texttt{P}'$, $\texttt{R}' \leftarrow \texttt{R}$, $\texttt{R} \leftarrow \texttt{SIB(R)}$; then $\texttt{SIB(R)} \leftarrow \texttt{Q}$, $\texttt{Q} \leftarrow \texttt{Q}'$. Set $\texttt{CHILD(P)} \leftarrow \texttt{Q}$ if $\texttt{P}=\texttt{P}'$, $\texttt{SIB(P)} \leftarrow \texttt{Q}$ if $\texttt{P}\neq\texttt{P}'$. Go to I8. Lutz

I5. [Case 1.] (Now component $\texttt{Q} = \texttt{R}$ is surrounded by $\texttt{P}' = \texttt{R}'$.) If $\texttt{P} = \texttt{P}'$, set $\texttt{CHILD(P)} \leftarrow \texttt{SIB(Q)}$; otherwise set $\texttt{SIB(P)} \leftarrow \texttt{SIB(Q)}$. Set $\texttt{R} \leftarrow \texttt{DORMANT(R}'\texttt{)}$. Then if $\texttt{R} = \Lambda$, set $\texttt{DORMANT(R}'\texttt{)} \leftarrow \texttt{SIB(Q)} \leftarrow \texttt{Q}$; otherwise $\texttt{SIB(Q)} \leftarrow \texttt{SIB(R)}$ and $\texttt{SIB(R)} \leftarrow \texttt{Q}$. Go to I8.

I6. [Case 2.] (Now $\texttt{Q}'$ is the parent of both $\texttt{P}'$ and $\texttt{R}$. Either $\texttt{P} = \texttt{P}'$ is childless, or $\texttt{P}$ is the last child of $\texttt{P}'$.) Absorb $\texttt{R}$ into $\texttt{P}'$ (see below). Set $\texttt{SIB(P}'\texttt{)} \leftarrow \texttt{SIB(R)}$ and $\texttt{R} \leftarrow \texttt{CHILD(R)}$. If $\texttt{P} = \texttt{P}'$, set $\texttt{CHILD(P)} \leftarrow \texttt{R}$; otherwise $\texttt{SIB(P)} \leftarrow \texttt{R}$. While $\texttt{R} \neq \Lambda$, set $\texttt{PARENT(R)} \leftarrow \texttt{P}'$ and $\texttt{R} \leftarrow \texttt{SIB(R)}$. Go to I8.

I7. [Case 3.] (Node $\texttt{P}' = \texttt{Q}$ is the last child of $\texttt{Q}' = \texttt{R}$, which is a child of $\texttt{R}'$.) Absorb $\texttt{P}'$ into $\texttt{R}'$ (see below). If $\texttt{P} = \texttt{P}'$, set $\texttt{P} \leftarrow \texttt{R}$. Otherwise set $\texttt{P}' \leftarrow \texttt{CHILD(P}'\texttt{)}$, and while $\texttt{P}' \neq \Lambda$ set $\texttt{PARENT(P}'\texttt{)} \leftarrow \texttt{R}'$, $\texttt{P}' \leftarrow \texttt{SIB(P}'\texttt{)}$; also set $\texttt{SIB(P)} \leftarrow \texttt{SIB(Q}'\texttt{)}$ and $\texttt{SIB(Q}'\texttt{)} \leftarrow \texttt{CHILD(Q)}$. If $\texttt{Q} = \texttt{CHILD(R)}$, set $\texttt{CHILD(R)} \leftarrow \Lambda$. Otherwise set $\texttt{R} \leftarrow \texttt{CHILD(R)}$, then $\texttt{R} \leftarrow \texttt{SIB(R)}$ until $\texttt{SIB(R)} = \texttt{Q}$, then $\texttt{SIB(R)} \leftarrow \Lambda$. Finally set $\texttt{P}' \leftarrow \texttt{R}'$.

I8. [Advance k.] Set $k \leftarrow k + 2$ and return to step I3.

I9. [Update the area.] Set $\texttt{AREA(P}'\texttt{)} \leftarrow \texttt{AREA(P}'\texttt{)} + \lceil a_j/2\rceil - \lceil a_{j-1}/2\rceil$. Then go back to I2 if $a_j = 2N + 3$.

I10. [Gobble up short a.] If $a_{j+1} \geq b_k$, go to I11. Otherwise set $\texttt{Q} \leftarrow \texttt{LOC(NODE(}t\texttt{))}$ and $t \leftarrow t + 1$. Set $\texttt{PARENT(Q)} \leftarrow \texttt{P}'$, $\texttt{DORMANT(Q)} \leftarrow \texttt{HEIR(Q)} \leftarrow \Lambda$; also $\texttt{ROW(Q)} \leftarrow s$, $\texttt{COL(Q)} \leftarrow \lceil a_j/2\rceil$, $\texttt{AREA(Q)} \leftarrow \lceil a_{j+1}/2\rceil - \lceil a_j/2\rceil$. If $\texttt{P} = \texttt{P}'$, set $\texttt{SIB(Q)} \leftarrow \texttt{CHILD(P)}$ and $\texttt{CHILD(P)} \leftarrow \texttt{Q}$; otherwise set $\texttt{SIB(Q)} \leftarrow \texttt{SIB(P)}$ and $\texttt{SIB(P)} \leftarrow \texttt{Q}$. Finally set $\texttt{P} \leftarrow \texttt{Q}$, $j \leftarrow j + 2$, and return to I3.

I11. [Move on.] Set $j \leftarrow j + 1$, $k \leftarrow k + 1$, $(\texttt{P},\texttt{P}') \leftarrow \text{next}(\texttt{P},\texttt{P}')$, and go to I3. ▌

To "absorb $\texttt{P}$ into $\texttt{Q}$" means to do the following things: If $(\texttt{ROW(P)},\texttt{COL(P)})$ is less than $(\texttt{ROW(Q)},\texttt{COL(Q)})$, set $(\texttt{ROW(Q)},\texttt{COL(Q)}) \leftarrow (\texttt{ROW(P)},\texttt{COL(P)})$. Set $\texttt{AREA(Q)} \leftarrow \texttt{AREA(P)} + \texttt{AREA(Q)}$. If $\texttt{DORMANT(Q)} = \Lambda$, set $\texttt{DORMANT(Q)} \leftarrow \texttt{DORMANT(P)}$; otherwise if $\texttt{DORMANT(P)} \neq \Lambda$, swap $\texttt{SIB(DORMANT(P))} \leftrightarrow \texttt{SIB(DORMANT(Q))}$. Finally, set $\texttt{HEIR(P)} \leftarrow \texttt{Q}$. (The $\texttt{HEIR}$ links could be used on a second pass to identify the final component of each pixel. Notice that the $\texttt{PARENT}$ links of dormant nodes are not kept up to date.)

[A similar algorithm was given by R. K. Lutz in *Comp. J.* **23** (1980), 262–269.]

180. Let $F(x,y) = x^2 - y^2 + 13$ and $Q(x,y) = F(x-\frac{1}{2}, y-\frac{1}{2}) = x^2 - y^2 - x + y + 13$. Apply Algorithm T to digitize the hyperbola from $(\xi,\eta) = (-6,7)$ to $(\xi',\eta') = (0,\sqrt{13})$; hence $x = -6$, $y = 7$, $x' = 0$, $y' = 4$. The resulting edges are $(-6,7)$ — $(-5,7)$ — $(-5,6)$ — $(-4,6)$ — $(-4,5)$ — $(-3,5)$ — $(-3,4)$ — $\cdots$ — $(0,4)$. Then apply it again with $\xi = 0$, $\eta = \sqrt{13}$, $\xi' = 6$, $\eta' = 7$, $x = 0$, $y = 4$, $x' = 6$, $y' = 7$; the same edges are found (in reverse order), but with negated x coordinates.

181. Subdivide at points (ξ,η) where $F_x(\xi,\eta) = 0$ or $F_y(\xi,\eta) = 0$, namely at the real roots of $\{Q(-(b\eta+d)/(2a), \eta+\frac{1}{2}) = 0,\ \xi = -(b\eta+d)/(2a) - \frac{1}{2}\}$ or the real roots of $\{Q(\xi+\frac{1}{2}, -(b\xi+e)/(2c)) = 0,\ \eta = -(b\xi+e)/(2c) - \frac{1}{2}\}$, if they exist.

182. By induction on $|x'-x| + |y'-y|$. Consider, for example, the case $x' < x$ and $y' > y$. We know from (iii) that (ξ,η) lies in the box $x - \frac{1}{2} \leq \xi < x + \frac{1}{2}$ and $y - \frac{1}{2} \leq \eta < y + \frac{1}{2}$, and from (ii) that the curve travels monotonically as it moves from (ξ,η) to (ξ',η'). It must therefore exit the box at the edge $(x-\frac{1}{2}, y-\frac{1}{2})$ — $(x-\frac{1}{2}, y+\frac{1}{2})$ or $(x-\frac{1}{2}, y+\frac{1}{2})$ — $(x+\frac{1}{2}, y+\frac{1}{2})$. The latter holds if and only if $F(x-\frac{1}{2}, y+\frac{1}{2}) < 0$, because the curve can't intersect that edge twice when $x' < x$. And $F(x-\frac{1}{2}, y+\frac{1}{2})$ is

the value $Q(x, y+1)$ that is tested in step T3, because of the initialization in step T1. (We assume that the curve doesn't go *exactly* through $(x-\frac{1}{2}, y+\frac{1}{2})$, by implicitly adding $\frac{1}{2}$ to the function F behind the scenes.)

183. Let $Q'(x,y) = -1 - Q(x,y)$. The key point is that $Q(x,y) < 0$ if and only if $Q'(x,y) \ge 0$. (Curiously the algorithm makes the same decisions, backwards, although it probes the values of Q' and Q in different places.)

184. Find a positive integer h so that $d = (\eta - \eta')h$ and $e = (\xi' - \xi)h$ are integers and $d+e$ is even. Then carry out Algorithm T with $x = \lfloor \xi + \frac{1}{2} \rfloor$, $y = \lfloor \eta + \frac{1}{2} \rfloor$, $x' = \lfloor \xi' + \frac{1}{2} \rfloor$, $y' = \lfloor \eta' + \frac{1}{2} \rfloor$, and $Q(x,y) = d(x - \frac{1}{2}) + e(y - \frac{1}{2}) + f$, where

$$f = \lfloor (\eta'\xi - \xi'\eta)h \rfloor - [d > 0 \text{ and } (\eta'\xi - \xi'\eta)h \text{ is an integer}].$$

(The '$d > 0$' term ensures that the opposite straight line, from (ξ', η') back to (ξ, η), will have precisely the same edges; see exercise 183.) Steps T1 and T6–T9 become much simpler than they were in the general case, because $R = d$ and $S = e$ are constant.

(F. G. Stockton [*CACM* **6** (1963), 161, 450] and J. E. Bresenham [*IBM Systems Journal* **4** (1965), 25–30] gave similar algorithms, but with diagonal edges permitted.)

185. (a) $B(\epsilon) = z_0 + 2\epsilon(z_1 - z_0) + O(\epsilon^2)$; $B(1-\epsilon) = z_2 - 2\epsilon(z_2 - z_1) + O(\epsilon^2)$.

(b) Every point of $S(z_0, z_1, z_2)$ is a convex combination of z_0, z_1, and z_2.

(c) Obviously true, since $(1-t)^2 + 2(1-t)t + t^2 = 1$.

(d) The collinear condition follows from (b). Otherwise, by (c), we need only consider the case $z_0 = 0$ and $z_2 - 2z_1 = 1$, where $z_1 = x_1 + iy_1$ and $y_1 \ne 0$. In that case all points lie on the parabola $4x = (y/y_1)^2 + 4yx_1/y_1$.

(e) Note that $B(u\theta) = (1-u)^2 z + 2u(1-u)((1-\theta)z_0 + \theta z_1) + u^2 B(\theta)$ for $0 \le u \le 1$.

[S. N. Bernshteĭn introduced $B_n(z_0, z_1, \ldots, z_n; t) = \sum_k \binom{n}{k}(1-t)^{n-k} t^k z_k$ in *Soobshchenii͡a Khar'kovskoe matematicheskoe obshchestvo* (2) **13** (1912), 1–2.]

186. We can assume that $z_0 = (x_0, y_0)$, $z_1 = (x_1, y_1)$, and $z_2 = (x_2, y_2)$, where the coordinates are (say) fixed-point numbers represented as 16-bit integers divided by 32.

If z_0, z_1, and z_2 are collinear, use the method of exercise 184 to draw a straight line from z_0 to z_2. (If z_1 doesn't lie between z_0 and z_2, the other edges will cancel out, because edges are implicitly XORed by a filling algorithm.) This case occurs if and only if $D = x_0y_1 + x_1y_2 + x_2y_0 - x_1y_0 - x_2y_1 - x_0y_2 = 0$.

Otherwise the points (x, y) of $S(z_0, z_1, z_2)$ satisfy $F(x, y) = 0$, where

$$F(x,y) = \bigl((x - x_0)(y_2 - 2y_1 + y_0) - (y - y_0)(x_2 - 2x_1 + x_0)\bigr)^2 \\ - 4D\bigl((x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0)\bigr)$$

and D is defined above. We multiply by 32^4 to obtain integer coefficients; then negate this formula and subtract 1, if $D < 0$, to satisfy condition (iv) of Algorithm T and the reverse-order condition. (See exercise 183.)

The monotonicity condition (ii) holds if and only if $(x_1 - x_0)(x_2 - x_1) > 0$ and $(y_1 - y_0)(y_2 - y_1) > 0$. If necessary, we can use the recurrence of exercise 185(e) to break $S(z_0, z_1, z_2)$ into at most three monotonic subsquines; for example, setting $\theta = (x_0 - x_1)/(x_0 - 2x_1 + x_2)$ will achieve monotonicity in x. (A slight rounding error may occur during this fixed point arithmetic, but the recurrence can be performed in such a way that the subsquines are definitely monotonic.)

Notes: When z_0, z_1, and z_2 are near each other, a simpler and faster method based on exercise 185(e) with $\theta = \frac{1}{2}$ is adequate for most practical purposes, if one doesn't care about making the exactly correct choice between local edge sequences like "up-then-left" versus "left-then-up." In the late 1980s, Sampo Kaasila chose to use squines

as the basic method of shape specification in the TrueType font format, because they can be digitized so rapidly. The METAFONT system achieves greater flexibility with cubic Bézier splines [see D. E. Knuth, *METAFONT: The Program* (Addison–Wesley, 1986)], but at the cost of extra processing time. A fairly fast "six-register algorithm" for the resulting cubic curves was, however, developed subsequently by John Hobby [*ACM Trans. on Graphics* **9** (1990), 262–277]. Vaughan Pratt introduced *conic splines*, which are sort of midway between squines and Bézier cubics, in *Computer Graphics* **9**, 3 (July 1985), 151–159. Conic spline segments can be elliptical and hyperbolic as well as parabolic, hence they require fewer intermediate points and control points than squines; furthermore, they can be handled by Algorithm T.

187. The following big-endian program assumes that $n \le 74880$.

```
         LOC  Data_Segment                 LDO  k,Initk
BITMAP   LOC  @+M*N/8                   0H SET  s,N/64
base     GREG @                         1H SET  a,h         A trick (see below)
GRAYMAP  LOC  @+M*N/64                     SET  r,8
GTAB     BYTE 255,252,249,246,243       2H LDOU t,base,k
         BYTE 240,236,233,230,227          MOR  u,c1,t
         BYTE 224,221,217,214,211          SUBU t,t,u       (Nypwise sums)
         BYTE 208,204,201,198,194          MOR  u,c2,t
         BYTE 191,188,184,181,178          AND  t,t,mu1
         BYTE 174,171,167,164,160          ADDU t,t,u       (Nybblewise sums)
         BYTE 157,153,150,146,142          MOR  u,c3,t
         BYTE 139,135,131,128,124          AND  t,t,mu2
         BYTE 120,116,112,108,104          ADDU t,t,u       (Bytewise sums)
         BYTE 100,96,92,88,84              ADDU a,a,t
         BYTE 79,75,70,66,61               INCL k,N/8       Move to next row.
         BYTE 56,52,46,41,36               SUB  r,r,1
         BYTE 30,24,18,10,0                PBNZ r,2B        Repeat 8 times.
Initk    OCTA BITMAP-GRAYMAP            3H SRU  t,a,56
corr     GREG N-8                          LDBU t,gtab,t
c1       GREG #4000100004000100            SLU  a,a,8
c2       GREG #2010000002010000            STBU t,z,0
c3       GREG #0804020100000000            INCL z,1
mu1      GREG #3333333333333333            PBN  a,3B        (The trick)
mu2      GREG #0f0f0f0f0f0f0f0f            SUB  k,k,corr
h        GREG #8080808080808080            SUB  s,s,1
gtab     GREG GTAB-#80                     PBNZ s,1B        Loop on columns.
         LOC  #100                         INCL k,7*N/8     Loop on groups
MakeGray LDA  z,GRAYMAP                    PBN  k,0B         of 8 rows. ▌
```

[Inspired by Neil Hunt's DVIPAGE, the author used such graymaps extensively when preparing new editions of *The Art of Computer Programming* in 1992–1998.]

188. If the rows of the bitmap are $(X_0, X_1, \ldots, X_{63})$, do the following operations for $k = 0, 1, \ldots, 5$: For all i such that $0 \le i < 64$ and $i \,\&\, 2^k = 0$, let $j = i + 2^k$ and either (a) set $t \leftarrow (X_i \oplus (X_j \gg 2^k)) \,\&\, \mu_{6,k}$, $X_i \leftarrow X_i \oplus t$, $X_j \leftarrow X_j \oplus (t \ll 2^k)$; or (b) set $t \leftarrow X_i \,\&\, \bar\mu_{6,k}$, $u \leftarrow X_j \,\&\, \mu_{6,k}$, $X_i \leftarrow ((X_i \ll 2^k) \,\&\, \bar\mu_{6,k}) \mid u$, $X_j \leftarrow ((X_j \gg 2^k) \,\&\, \mu_{6,k}) \mid t$.

[The basic idea is to transform $2^k \times 2^k$ submatrices for increasing k, as in exercise 5–12. Speedups are possible with `MMIX`, using `MOR` and `MUX` as in exercise 205, and using `LDTU`/`STTU` when $k = 5$. See L. J. Guibas and J. Stolfi, *ACM Transactions on Graphics*

TrueType
Bézier
Knuth
METAFONT
six-register algorithm
Hobby
Pratt
conic splines
elliptical
hyperbolic
big-endian
Hunt
DVIPAGE
Knuth
trick
MOR
magic
`MOR`
`MUX`
Guibas
Stolfi

1 (1982), 204–207. Incidentally, Theorem P and answer 54 show that $\Omega(n \log n)$ operations on n-bit numbers are needed to transpose an $n \times n$ bit matrix. An application that needs frequent transpositions might therefore be better off using a redundant representation, maintaining its matrices in both normal and transposed form.]

lower bounds
redundant representation
pigeonhole principle
continuant polynomial
Chebyshev polynomial

189. (a) We must have $\alpha_{j+1} = f(\alpha_j) \oplus \alpha_{j-1}$ for $j \geq 1$, where $\alpha_0 = 0\ldots0$ and $f(\alpha) = ((\alpha \ll 1)\,\&\,1\ldots1) \oplus \alpha \oplus (\alpha \gg 1)$. The elements of the bottom row α_m satisfy the parity condition if and only if this rule makes α_{m+1} entirely zero.

(b) True. The parity condition on matrix entries a_{ij} is $a_{ij} = a_{(i-1)j} \oplus a_{i(j-1)} \oplus a_{i(j+1)} \oplus a_{(i+1)j}$, where $a_{ij} = 0$ if $i = 0$ or $i = m+1$ or $j = 0$ or $j = n+1$. If two matrices (a_{ij}) and (b_{ij}) satisfy this condition, so does (c_{ij}) when $c_{ij} = a_{ij} \oplus b_{ij}$.

(c) The upper left submatrix consisting of all rows that precede the first all-zero row (if any) and all columns that precede the first all-zero column (if any) is perfect. And this submatrix determines the entire matrix, because the pattern on the other side of a row or column of zeros is the top/bottom or left/right reflection of its neighbor. For example, if $\alpha_{m'+1}$ is zero, then $\alpha_{m'+1+j} = \alpha_{m'+1-j}$ for $1 \leq j < m - m'$.

(d) Starting with a given vector α_1 and using the rule in (a) will always lead to a row with $\alpha_{m+1} = 0\ldots0$. Proof: We must have $(\alpha_j, \alpha_{j+1}) = (\alpha_k, \alpha_{k+1})$ for some $0 \leq j < k \leq 2^{2n}$, by the pigeonhole principle. If $j > 0$ we also have $(\alpha_{j-1}, \alpha_j) = (\alpha_{k-1}, \alpha_k)$, because $\alpha_{j-1} = f(\alpha_j) \oplus \alpha_{j+1} = f(\alpha_k) \oplus \alpha_{k+1} = \alpha_{k-1}$. Therefore the first repeated pair begins with a row α_k of zeros. Furthermore we have $\alpha_i = \alpha_{k-i}$ for $0 \leq i \leq k$; hence the first all-zero row α_{m+1} occurs when m is $k-1$ or $k/2 - 1$.

Rows $\alpha_1, \ldots, \alpha_m$ will form a perfect pattern unless there is a column of 0s. There are $t > 0$ such columns if and only if $t+1$ is a divisor of $n+1$ and α_1 has the form $\alpha 0 \alpha^R 0 \ldots 0 \alpha$ (t even) or $\alpha 0 \alpha^R 0 \ldots 0 \alpha^R$ (t odd), where $|\alpha| + 1 = (n+1)/(t+1)$.

(e) This starting vector does not have the form forbidden in (d).

190. (a) The former is $\alpha_1, \alpha_2, \ldots$ if and only if the latter is $0\alpha_1 0\alpha_1^R$, $0\alpha_2 0\alpha_2^R$, ...

(b) Let the binary string $a_0 a_1 \ldots a_{N-1}$ correspond to the polynomial $a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$, and let $y = x^{-1}{+}1{+}x$. Then $\alpha_0 = 0\ldots0$ corresponds to $F_0(y)$; $\alpha_1 = 10\ldots0$ corresponds to $F_1(y)$; and by induction α_j corresponds to $F_j(y)$, mod $x^N + 1$ and mod 2. For example, when $N = 6$ we have $\alpha_2 = 110001 \leftrightarrow 1 + x + x^5$ because $x^{-1} \bmod (x^6 + 1) = x^5$, etc.

(c) Again, induction on j.

(d) The identity in the hint holds by induction on m, because it is clearly true when $m = 1$ and $m = 2$. Working mod 2, this identity yields the simple equations

$$F_{2k}(y) = yF_k(y)^2; \qquad F_{2k-1}(y) = (F_{k-1}(y) + F_k(y))^2.$$

So we can go from the pair $P_k = (F_{k-1}(y) \bmod (x^N{+}1), F_k(y) \bmod (x^N{+}1))$ to the pair P_{k+1} in $O(n)$ steps, and to the pair P_{2k} in $O(n^2)$ steps. We can therefore compute $F_j(y) \bmod (x^N + 1)$ after $O(\log j)$ iterations. Multiplying by $f_\alpha(x) + f_\alpha(x^{-1})$ and reducing mod $x^N + 1$ then allows us to read off the value of α_j.

Incidentally, $F_{n+1}(x)$ is the special case $K_n(x, x, \ldots, x)$ of a continuant polynomial; see Eq. 4.5.3–(4). We have $F_{n+1}(x) = \sum_{k=0}^{n} \binom{n-k}{k} x^{n-2k} = i^{-n} U_n(ix/2)$, where U_n is the classical Chebyshev polynomial defined by $U_n(\cos\theta) = \sin((n+1)\theta)/\sin\theta$.

191. (a) By exercise 190(c), $c(q)$ is the least $j > 0$ such that $(x{+}x^{-1})F_j(x^{-1}{+}1{+}x) \equiv 0$ (modulo $x^{2q} + 1$), using polynomial arithmetic mod 2. Equivalently, it's the smallest positive j for which $F_j(y)$ is a multiple of $(x^{2q} + 1)/(x^2 + 1) = (1 + x + \cdots + x^{q-1})^2$, when $y = x^{-1}{+}1{+}x$.

finite field
Lights Out
Uri
Meerö
XL25
Singmaster
Gebhardt
Sutner

(b) Use the method of exercise 190(d) to evaluate $((x+x^{-1})F_j(y)) \bmod (x^{2q}+1)$ when $j = M/p$, for all prime divisors p of M. If the result is zero, set $M \leftarrow M/p$ and repeat the process. If no such result is zero, $c(q) = M$.

(c) We want to show that $c(2^e)$ is a divisor of $3 \cdot 2^{e-1}$ but not of $3 \cdot 2^{e-2}$ or 2^{e-1}. The latter holds because $F_{2^{e-1}}(y) = y^{2^{e-1}-1}$ is relatively prime to $x^{2^{e+1}}+1$. The former holds because

$$F_{3\cdot 2^{e-1}}(y) = y^{2^{e-1}-1}F_3(y)^{2^{e-1}} = y^{2^{e-1}-1}(1+y)^{2^e} = y^{2^{e-1}-1}(x^{-1}+x)^{2^e},$$

which is $\equiv 0$ modulo $x^{2^{e+1}}+1$ but not modulo $x^{2^e}+1$.

(d) $F_{2^e-1}(y) = \sum_{k=1}^{e} y^{2^e-2^k}$. Since $y = x^{-1}(1+x+x^2)$ is relatively prime to x^q+1, we have $y^{-1} \equiv a_0 + a_1 x + \cdots + a_{q-1}x^{q-1}$ (modulo x^q+1) for some coefficients a_i; hence

$$y^{-2^k} \equiv a_0 + a_1 x^{2^k} + \cdots + a_{q-1}x^{2^k(q-1)} \equiv a_0 + a_1 x^{2^{k+e}} + \cdots + a_{q-1}x^{2^{k+e}(q-1)} \equiv y^{-2^{k+e}}$$

(modulo x^q+1) for $0 \le k < e$, and it follows that $F_{2^{2e}-1}(y)$ is a multiple of $x^{2q}+1$.

(e) In this case $c(q)$ divides $4(2^{2e}-1)$. Proof: Let $x^q+1 = f_1(x)f_2(x)\ldots f_r(x)$ where $f_1(x) = x+1$, $f_2(x) = x^2+x+1$, and each $f_i(x)$ is irreducible mod 2. Since q is odd, these factors are distinct. Therefore, in the finite field of polynomials mod $f_j(x)$ for $j \ge 3$, we have $y^{-2^k} = y^{-2^{k+e}}$ as in (d). Consequently $F_{2^{2e}-1}(y)$ is a multiple of $f_3(x)\ldots f_r(x) = (x^q+1)/(x^3+1)$. So $F_{2(2^{2e}-1)}(y) = y^3 F_{2^{2e}-1}(y)^2$ is a multiple of $(x^{2q}+1)/(x^2+1)$ as desired.

(f) If $F_{c(q)}(y)$ is a multiple of $x^{2q}+1$, it's easy to see that $c(2q) = 2c(q)$. Otherwise $F_{3c(q)}(y)$ is a multiple of $F_3(y) = (1+y)^2 = x^{-2}(1+x)^4$; hence $F_{6c(q)}(y)$ is a multiple of $x^{4q}+1$ and $c(2q)$ divides $6c(q)$. The latter case can happen only when q is odd.

Notes: Parity patterns are related to a popular puzzle called "Lights Out," which was invented in the early 1980s by Dario Uri, also invented independently about the same time by László Meerö and called XL25. [See David Singmaster's *Cubic Circular*, issues 7&8 (Summer 1985), 39–42; Dieter Gebhardt, *Cubism For Fun* **69** (March 2006), 23–25.] Klaus Sutner has pursued further aspects of this theory in *Theoretical Computer Science* **230** (2000), 49–73.

192. Let $b_{(2i)(2j)} = a_{ij}$, $b_{(2i+1)(2j)} = a_{ij} \oplus a_{(i+1)j}$, $b_{(2i)(2j+1)} = a_{ij} \oplus a_{i(j+1)}$, and $b_{(2i+1)(2j+1)} = 0$, for $0 \le i \le m$ and $0 \le j \le n$, where we regard $a_{ij} = 0$ when $i = 0$ or $i = m+1$ or $j = 0$ or $j = n+1$. We don't have $(b_{(2i)1}, b_{(2i)2}, \ldots, b_{(2i)(2n+1)}) = (0,0,\ldots,0)$ because $(a_{i1},\ldots,a_{in}) \ne (0,\ldots,0)$ for $1 \le i \le m$. And we don't have $(b_{(2i+1)1}, b_{(2i+1)2}, \ldots, b_{(2i+1)(2n+1)}) = (0,0,\ldots,0)$ because adjacent rows $(a_{i1},\ldots,a_{in})$ and $(a_{(i+1)1},\ldots,a_{(i+1)n})$ always differ for $0 \le i \le m$ when m is odd.

193. Set $\beta_i \leftarrow (1 \ll (n-i)) \mid (1 \ll (i-1))$ for $1 \le i \le m$, where $m = \lceil n/2 \rceil$. Also set $\gamma_i \leftarrow (\beta_1 \,\&\, \alpha_{i1}) + (\beta_2 \,\&\, \alpha_{i2}) + \cdots + (\beta_m \,\&\, \alpha_{im})$, where α_{ij} is the jth row of the parity pattern that begins with β_i; vector γ_i records the diagonal elements of such a matrix. Then set $r \leftarrow 0$ and apply subroutine N of answer 194 for $i \leftarrow 1, 2, \ldots, m$. The resulting vectors $\theta_1, \ldots, \theta_r$ are a basis for all $n \times n$ parity patterns with 8-fold symmetry.

To test if any such pattern is perfect, let the pattern starting with θ_i first be zero in row c_i. If any $c_i = n+1$, the answer is yes. If $\mathrm{lcm}(c_1,\ldots,c_r) < n$, the answer is no. If neither of these conditions decides the matter, we can resort to brute-force examination of $2^r - 1$ nonzero linear combinations of the θ vectors.

For example, when $n = 9$ we find $\gamma_1 = 111101111$, $\gamma_2 = \gamma_3 = 010101010$, $\gamma_4 = 000000000$, $\gamma_5 = 001010100$; then $r = 0$, $\theta_1 = 011000110$, $\theta_2 = 000101000$, $c_1 = c_2 = 5$. So there is no perfect solution.

In the author's experiments for $n \le 3000$, "brute force" was needed only when $n = 1709$. Then $r = 21$ and the values of c_i were all equal to 171 or 855 except that $c_{21} = 342$. The solution $\theta_1 \oplus \theta_{21}$ was found immediately.

The answers for $1 \le n \le 383$ are 4, 5, 11, 16, 23, 29, 30, 32, 47, 59, 62, 64, 65, 84, 95, 101, 119, 125, 126, 128, 131, 154, 164, 170, 185, 191, 203, 204, 239, 251, 254, 256, 257, 263, 314, 329, 340, 341, 371, 383.

[A fractal similar to Fig. 20, called the "mikado pattern," appears in a paper by H. Eriksson, K. Eriksson, and J. Sjöstrand, *Advances in Applied Math.* **27** (2001), 365. See also S. Wolfram, *A New Kind of Science* (2002), rule 150R on page 439.]

194. Set $\beta_i \leftarrow 1 \ll (m-i)$ and $\gamma_i \leftarrow \alpha_i$ for $1 \le i \le m$; also set $r \leftarrow 0$. Then perform the following subroutine for $i = 1, 2, \ldots, m$:

N1. [Extract low bit.] Set $x \leftarrow \gamma_i \mathbin{\&} -\gamma_i$. If $x = 0$, go to N4.

N2. [Find j.] Find the smallest $j \ge 1$ such that $\gamma_j \mathbin{\&} x \ne 0$ and $\gamma_j \mathbin{\&} (x-1) = 0$.

N3. [Dependent?] If $j < i$, set $\gamma_i \leftarrow \gamma_i \oplus \gamma_j$, $\beta_i \leftarrow \beta_i \oplus \beta_j$, and return to N1. (These operations preserve the matrix equation $C = BA$.) Otherwise terminate the subroutine (because γ_i is linearly independent from $\gamma_1, \ldots, \gamma_{i-1}$).

N4. [Record a solution.] Set $r \leftarrow r+1$ and $\theta_r \leftarrow \beta_i$. ▮

At the conclusion, the $m - r$ nonzero vectors γ_i are a basis for the vector space of all linear combinations of $\alpha_1, \ldots, \alpha_m$; they're characterized by their low bits.

195. (a) #`0a`; #`cea3`; #`e7ae97`; #`f09d8581`.

(b) If $\lambda x = \lambda x'$, the result is clear because $l = l'$. Otherwise we have $\alpha_1 < \alpha'_1$.

(c) Set $j \leftarrow k$; while $\alpha_j \oplus$ #`80` $<$ #`40`, set $j \leftarrow j-1$. Then $\alpha(x^{(i)})$ begins with α_j.

196. (a) #`000a`; #`03a3`; #`7b97`; #`d834dd41`.

(b) Lexicographic order is *not* preserved when, say, $x =$ #`ffff` and $x' =$ #`10000`.

(c) To answer this question properly one needs to know that the 2048 integers in the range #`d800` $\le x <$ #`e000` are not legal codepoints of UCS; they are called *surrogates*. With this understanding, $\beta(x^{(i)})$ begins at α_k if $\alpha \oplus$ #`dc00` $\ge$ #`0400`, otherwise it begins at α_{k-1}.

197. $a =$ #`e5000000`, $b = 3$, $c =$ #`fe`. (We could let $b = 0$, but then a would be huge. This trick was suggested by P. Raynaud-Richard in 1997.)

198. We want $\alpha_1 >$ #`c1`; $2^8\alpha_1 + \alpha_2 <$ #`f490`; and either $(\alpha_1 \mathbin{\&} -\alpha_1) + \alpha_1 <$ #`100` or $\alpha_1 + \alpha_2 >$ #`17f`. These conditions hold if and only if

$$(\#\mathtt{c1} - \alpha_1) \mathbin{\&} (2^8\alpha_1 + \alpha_2 - \#\mathtt{f490}) \mathbin{\&} (((\alpha_1 \mathbin{\&} -\alpha_1) + \alpha_1 - \#\mathtt{100}) \mid (\#\mathtt{17f} - \alpha_1 - \alpha_2)) < 0.$$

Markus Kuhn suggests adding the further clause '$\mathbin{\&} (\#\mathtt{20} - ((2^8\alpha_1 + \alpha_2) \oplus \#\mathtt{eda}))$', to ensure that $\alpha_1\alpha_2$ doesn't begin the encoding of a surrogate.

199. If $\$0 = (x_7 \ldots x_1x_0)_{256}$ then $\$3 = S_2(x_7, x_4, x_2) = (x_7 \mathbin{\&} x_4) \mid (x_7 \mathbin{\&} x_2) \mid (x_4 \mathbin{\&} x_2)$.

200. `MOR x,c,x`, where `c` = #`f0f0f0f00f0f0f0f`.

201. `MOR x,x,c`, where `c` = #`c0c030300c0c0303`; then `MOR x,mone,x`. (See answer 206.)

202. `a` = #`0008000400020001`, `b` = #`0f0f0f0f0f0f0f0f`, `c` = #`0606060606060606`, `d` = #`0000002700000000`, `e` = #`2a2a2a2a2a2a2a2a`. (The ASCII code for `0` is $6 +$ #`2a`; the ASCII code for `a` is $6 +$ #`2a` $+ 10 +$ #`27`.)

203. $p =$ #`8008400420021001`, $q =$ #`8020080240100401` (the transpose of p), $r =$ #`4080102004080102` (a symmetric matrix), and $m =$ #`aa55aa55aa55aa55`.

Knuth
mikado pattern
Eriksson
Eriksson
Sjöstrand
Wolfram
vector space
surrogates
Raynaud-Richard
Kuhn
surrogate

Warren
swap
MUX
Boolean matrix product
multilinear representation
MXOR
big-endian
GULLIVER
SWIFT

204. Just change p to `#0880044002200110`. (Incidentally, these shuffles can also be defined as permutations on $z = (z_{63} \ldots z_1 z_0)_2$ in another way: The outshuffle maps $z_j \mapsto z_{(2j) \bmod 63}$ while the inshuffle maps $z_j \mapsto z_{(2j+1) \bmod 65}$.)

205. (Solution by H. S. Warren, Jr.) The text's 7-swap, 14-swap, 28-swap method can be implemented with only 12 instructions:

```
MOR t,x,c1;  MOR t,c1,t;  PUT rM,m1;  MUX y,x,t;
MOR t,y,c2;  MOR t,c2,t;  PUT rM,m2;  MUX y,y,t;
MOR t,y,c3;  MOR t,c3,t;  PUT rM,m3;  MUX y,y,t;
```

here `c1` = `#4080102004080102`, `c2` = `#2010804002010804`, `c3` = `#0804020180402010`, `m1` = `#aa55aa55aa55aa55`, `m2` = `#cccc3333cccc3333`, `m3` = `#f0f0f0f00f0f0f0f`.

206. Four instructions suffice: `MXOR y,p,x; MXOR x,mone,x; MXOR x,x,q; XOR x,x,y;` here `p` = `#80c0e0f0f8fcfeff`, `mone` $= -1$, and `q` $= \bar{\texttt{p}}$.

207. `SLU x,one,x; MOR x,b,x; AND x,x,a; MOR x,x,#ff;` here register `one` $= 1$.

208. In general, element ij of the Boolean matrix product AXB is $\bigvee\{x_{kl} \mid a_{ik} \wedge b_{lj}\}$. For this problem we choose $a_{ik} = [i \subseteq k]$ and $b_{lj} = [l \supseteq j]$; the answer is '`MOR t,f,a; MOR t,b,t`' where `a` = `#80c0a0f088ccaaff` and `b` = `#ff5533110f050301` $= \texttt{a}^T$.

(Notice that this trick gives a simple test $[f = \hat{f}]$ for monotonicity. Furthermore, the 64-bit result $(t_{63} \ldots t_1 t_0)_2$ gives the coefficients of the multilinear representation

$$f(x_1, \ldots, x_6) = (t_{63} + t_{62}x_6 + \cdots + t_1 x_1 x_2 x_3 x_4 x_5 + t_0 x_1 x_2 x_3 x_4 x_5 x_6) \bmod 2,$$

if we substitute `MXOR` for `MOR`, by the result of exercise 7.1.1–11.)

209. If $\cdot$ denotes `MXOR` as in (183) and $b = (\beta_7 \ldots \beta_1 \beta_0)_{256}$ has bytes β_j, we can evaluate

$$c = (a \cdot B_0^L) \oplus ((a \ll 8) \cdot (B_1^L + B_0^U)) \oplus ((a \ll 16) \cdot (B_2^L + B_1^U)) \oplus \cdots \oplus ((a \ll 56) \cdot (B_7^L + B_6^U)),$$

where $B_j^U = (q\beta_j) \mathbin{\&} m$, $B_j^L = (((q\beta_j) \ll 8) + \beta_j) \mathbin{\&} \overline{m}$, $q =$ `#0080402010080402`, and $m =$ `#7f3f1f0f07030100`. (Here $q\beta_j$ denotes *ordinary* multiplication of integers.)

210. In this big-endian computation, register `nn` holds $-n$, and register `data` points to the octabyte following the given bytes $\alpha_{n-1} \ldots \alpha_1 \alpha_0$ in memory (with α_{n-1} first). The constants `aa` = `#8381808080402010` and `bb` = `#339bcf6530180c06` correspond to matrices A and B, found by computing the remainders $x^k \bmod p(x)$ for $72 \le k < 80$.

	`SET`	`c,0`	$c \leftarrow 0$.	`LDOU`	`t,data,nn`	$t \leftarrow$ next octa.
	`ADD`	`nn,nn,8`	$n \leftarrow n - 8$.	`XOR`	`u,u,c`	$u \leftarrow u \oplus c$.
	`LDOU`	`t,data,nn`	$t \leftarrow$ next octa.	`SLU`	`c,v,56`	$c \leftarrow v \ll 56$.
	`BZ`	`nn,2F`	Done if $n = 0$.	`SRU`	`v,v,8`	$v \leftarrow v \gg 8$.
1H	`MXOR`	`u,aa,t`	$u \leftarrow t \cdot A$.	`XOR`	`u,u,v`	$u \leftarrow u \oplus v$.
	`MXOR`	`v,bb,t`	$v \leftarrow t \cdot B$.	`XOR`	`t,t,u`	$t \leftarrow t \oplus u$.
	`ADD`	`nn,nn,8`	$n \leftarrow n - 8$.	`PBN`	`nn,1B`	Repeat if $n > 0$. ▌

A similar method finishes the job, with no auxiliary table needed:

2H	`SET`	`nn,8`	$n \leftarrow 8$.	`SRU`	`v,v,8`	$v \leftarrow v \gg 8$.
3H	`AND`	`x,t,ffooo`	$x \leftarrow$ high byte.	`XOR`	`t,t,v`	$t \leftarrow t \oplus v$.
	`MXOR`	`u,aaa,x`	$u \leftarrow x \cdot A'$.	`SUB`	`nn,nn,1`	$n \leftarrow n - 1$.
	`MXOR`	`v,bbb,x`	$v \leftarrow x \cdot B'$.	`PBP`	`nn,3B`	Repeat if $n > 0$.
	`SLU`	`t,t,8`	$t \leftarrow t \ll 8$.	`XOR`	`t,t,c`	$t \leftarrow t \oplus c$.
	`XOR`	`t,t,u`	$t \leftarrow t \oplus u$.	`SRU`	`crc,t,48`	Return $t \gg 48$. ▌

Here `aaa` = `#8381808080808080`, `bbb` = `#0383c363331b0f05`, and `ffooo` = `#ff00...00`.

The Books of the Big-Endians *have been long forbidden.*
— LEMUEL GULLIVER, *Travels Into Several Remote Nations of the World* (1726)

211. By considering the irreducible factors of the characteristic polynomial of X, we must have $X^n = I$ where $n = 2^3 \cdot 3^2 \cdot 5 \cdot 7 \cdot 17 \cdot 31 \cdot 127 = 168661080$. Neill Clift has shown that $l(n-1) = 33$ and found the following sequence of 33 `MXOR` instructions to compute $Y = X^{-1} = X^{n-1}$: `MXOR t,x,x;` `MXOR $1,t,x;` `MXOR $2,t,$1;` `MXOR $3,$2,$2;` `MXOR t,$3,$3;` S^6; `MXOR t,t,$2;` S^3; `MXOR $1,t,$1;` `MXOR t,$1,$3;` S^{13}; `MXOR t,t,$1;` S; `MXOR y,t,x;` here S stands for '`MXOR t,t,t`'. To test if X is nonsingular, do `MXOR t,y,x` and compare `t` to the identity matrix `#8040201008040201`.

Clift
`MXOR`
identity matrix

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

0–1 matrices, 67–70, *see also* Bitmaps.
  multiplication of, 50–51, 56.
  transposing, 15, 56, 67, 69, 80.
  triangularizing, 68.
0–1 principle, 54.
−1 (the constant $(\cdots 111)_2$), 3, 8, 9, 50, 71, 76, 106, 107.
2-adic chains, 23–27, 37, 61, 91, 96.
2-adic fractions, 9, 75.
2-adic integers, 2, 16, 21, 53, 55.
  as a metric space, 74.
2-bit encoding for 3-state data, 28–31, 63.
2-cube equivalence, 29–30.
2-dimensional data allocation, 16.
`2ADDU` (times 2 and add unsigned), 79, 84.
3-valued logic, 31, 63.
4-neighbors, *see* Rook-neighbors, 40.
`4ADDU` (times 4 and add unsigned), 79.
8-neighbors, *see* King-neighbors, 40.
`8ADDU` (times 8 and add unsigned), 79.
`16ADDU` (times 16 and add unsigned), 79.
∞ (infinity), 8, 55.
δ-maps, 84.
δ-shifts, 16.
δ-swaps, 13–15, 50, 55–56, 107.
λx ($\lfloor \lg x \rfloor$), *see* Binary logarithm.
μ (average memory access time), 117.
μ_k and $\mu_{d,k}$, *see* Magic masks.
νx, *see* Sideways addition.
ρx, *see* Ruler function.
υ (instruction cycle time), 117.

Absorption laws, 3.
Abstract RISC (reduced-instruction-set computer) model, 26.
Ackland, Bryan David, 44.
Acyclic digraph, 33.
Addition, 3, 19.
  bytewise, 19, 87.
  scattered, 18, 57.
  sideways, 2, 11–12, 55, 62, 94.
Adjacency matrix of a graph, 28, 62.
Adventure game, 85.
Agrawal, Dharma Prakash (धर्म प्रकाश अग्रवाल), 71.
Albers, Susanne, 88.
Allocation of memory, 22, 54, 59.
Allouche, Jean-Paul, 78.
Alpha channels, 59.
Alphabetic data, 20, 59.
Analysis of algorithms, 55, 85.
Ancestors in a forest, 33.
  nearest common, 33–35, 64.
AND (bitwise conjunction), 2–3.
Animating functions, 53, 56.
Ariyoshi, Hiromu (有吉弘), 92.
Arndt, Jörg Uwe, 76, 84.
ASCII: American Standard Code for Information Interchange, iv, 59, 69, 117.
Associative laws, 3, 72.
Asterisk codes for subcubes, 18, 63.
Averages, bytewise, 19, 59.

Background of an image, 42.
Balanced branching functions, 53.
Balanced ternary notation, 63, 79.
Banyan networks, 81.
Basic RAM (random-access machine) model, 26, 62, 91.
Baumgart, Bruce Guenther, 12.
Bays, John Carter, 77.
`BDIF` (byte difference), 20, 86–87.
Beneš, Václav Edvard, 13.
Bentley, Jon Louis, 95.
Berlekamp, Elwyn Ralph, 21, 73, 98.
Bernshteĭn, Sergeĭ Natanovich (Бернштейн, Сергей Натанович), 102.
BESM-6 (БЕСМ-6) computer, 83.
Beyer, Wendell Terry, 42.
Bézier, Pierre Etienne, splines, 48, 66–67, 103.
Big-endian convention, 6–8, 12, 20, 77, 103, 107.
Binary basis, 71.
Binary logarithm ($\lambda x = \lfloor \lg x \rfloor$), 10–11, 21, 25, 33, 55, 60–61, 64.
Binary recurrence relations, 8, 10, 55.
Binary search trees, 64, 79.
Binary tree structures, 32.
Binary valuation, *see* Ruler function.
Binary-coded decimal notation, 60.
Bipartite graphs, 14–15, 97.
Bit boards, 32, 63.
Bit codes for subcubes, 18, 63.
Bit permutations, 13–17, 25, 50.
Bitmaps, 39–48, 64–68.
  cleaning, 65.
  drawing on, 48.
  filling contours in, 44–48, 66–67.
  rotation and transposition of, 67.
Bitwise manipulations, 1–108.
Black pixels, 4, 40, 67.
Bolyai, János, 36.
Bookworm problem, 54.
Boolean matrices, 50, 69, *see also* Bitmaps.
  multiplication of, 50, 56, 107.

Borkowski, Ludwik Stefan, 31.
Borrows, 86–87.
Boundary curves, digitized, 44–48.
Bouton, Charles Leonard, 71.
Branch instructions, 10, 48–49, 61.
Branching functions, 53, 56.
Branchless computation, 23–26, 48–49, 69, 70.
Braymore, Caroline, 1.
Breadth-first search, 91, 97.
Brent, Richard Peirce, 83.
Bresenham, Jack Elton, 102.
Breuer, Melvin Allen, 94.
Broadword chains, 23–27, 60–65, 96.
  strong, 61.
Broadword computations, 21–27, 60–65.
Brodal, Gerth Stølting, 22.
Brodnik, Andrej, 27.
Bron, Coenraad, 92.
Brooker, Ralph Anthony, 2.
Brown, David Trent, 51.
Bruijn, Nicolaas Govert de, cycles, 10.
Büchi, Julius Richard, 75.
Butterfly networks, 56.
Byte: An 8-bit quantity, 69.
Byte permutations, 50.
Bytes, parallel operations on, *see* Multibyte processing.

Cache memory, 5, 35, 49, 77, 91.
Cahn, Leonard, 40.
Cancellation law, 72.
Cantor, Georg Ferdinand Ludwig Philipp, 85.
Cardinality of a set, 11.
Carries, 18–19, 25, 86–87.
Cartesian coordinates, 44.
Cartesian trees, 79, 95.
Cellular automata, 40–43, 65–66.
Chebyshev (= Tschebyscheff), Pafnutii Lvovich (Чебышев, Пафнутий Львович), polynomials, 104.
Cheshire cat, 42–43, 65, 66, 100.
Chessboards, 32, 63.
Chung, Kin-Man (鍾建民), 17, 58.
Circles, digitized, 44, 47.
Circular lists, 62, 100.
Cleaning images, 65, 98.
Clift, Neill Michael, 108.
Cliques, maximal, 62–63.
Closed bitmaps, 65.
Collation of bits, 2.
Colman, George, the younger, 1.
Combinations, 75–76.
Commutative laws, 3, 71.
Comparison of bytes, 21, 60.
Complementation, 3, 52, 92.
Complete binary trees, 33, 74.
  infinite, 53.
Composition of permutations, 53, 56–57.
Compression of scattered bits, 16, 57, 83.
Conditional-set instructions, 9–10, 48.
Conic sections, digitizing, 44–48, 66–67.
Conic splines, 103.
Conjunction, in 3-valued logic, 31.
Connectivity structure of an image, 41–43, 66.
Consensus of subcubes, 63.
Continuant polynomials, 104.
Control points, 48.
Convex optimization, 85.
Conway, John Horton, 40, 73, 74, 98.
  field, 52.
CRC (cyclic redundancy check), 51, 70.
Crossbar modules, 14, 58.
`CSNZ` (conditional set if nonzero), 10, 48–49, 88.
`CSOD` (conditional set if odd), 79.
`CSZ` (conditional set if zero), 9, 77, 78.
Custering, 39, 44, 64–65.
Cycles in a graph, 15.
Cyclic redundancy checking, 51, 70.
Cyclic shifts, 17, 56, 86.
Cylinder, hyperbolic, 39, 97.

Dallos, József, 9.
Dates, packed, 4, 60.
de Bruijn, Nicolaas Govert, cycles, 10.
Depth of a Boolean function, 13.
Descartes, René, coordinates, 44.
Dietz, Henry Gordon, 19, 85.
Digitization of contours, 44–48, 66–67.
Dijkstra, Edsger Wybe, 85.
Dirichlet, Johann Peter Gustav Lejeune, generating function, 78.
Discrete logarithm, *see* Binary logarithm.
Disjointness testing, 58.
Disjunction, in 3-valued logic, 31.
Distance between 2-adic integers, 74.
Distinct bytes, testing for, 59.
Distribution networks, *see* Mapping networks.
Distributive laws, 3, 72.
Divide and conquer paradigm, 12, 16.
Division, 54.
  avoiding, 4, 54.
  by 10, 24.
  by powers of 2, 3–4.
  in Conway's field, 52.
  of 2-bit numbers, 59.
Dominating sets, minimum, 98.
Don't-cares, 18, 29–30, 81, 94.
Dot-minus operation ($x \dot{-} y$), v, 20, 24, 61, 82, 96.
Double order for traversing trees, 100–101.
Drawing on a bitmap, 48.
Duality between 0 and 1, 99.
Duguid, Andrew Melville, 13.
DVIPAGE program, 103.

Edges between pixels, 44–48, 66–67.
EDSAC computer, 2, 11.
Eight queens problem, 92.
Ellipses, 44–47, 103.
Encoding of ternary data, 28–31, 63.
Eofill (even/odd filling), 47.
Equality of bytes, 20, 59, 60.
Equivalence, in 3-valued logic, 63.
Eratosthenes of Cyrene (Ἐρατοσθένης ὁ Κυρηναῖος); sieve (κόσκινον), 5, 54.
Eriksson, Henrik, 106.
Eriksson, Kimmo, 106.
Escher, George A., 37.
Escher, Maurits Cornelis, 37.
Euclid (Εὐκλείδης), 36.
Extracting bits, 2, 4, 8.
  and compressing them, 16, 57, 83.
  the least significant only ($2^{\rho x}$), 8–10, 18, 54.
  the most significant only ($2^{\lambda x}$), 11, 60–62, 89.

Fast Fourier transforms, 56.
Ferranti Mercury computer, 2.
Fibonacci, Leonardo, of Pisa (= Leonardo filio Bonacii Pisano), numbers, 36.
Fibonacci number system, 36, 64; *see also* NegaFibonacci number system.
  odd, 96.
Fibonacci polynomials, 67–68.
Fields, algebraic, 50, 52, 105.
Fields of data, *see* Packing of data.
Filling a contour in a bitmap, 44–48, 66–67.
Fingerprints, 40.
Finite fields, 50, 105.
Finite state automata, 89.
Fischer, Johannes Christian, 95.
Fisher, Randall James, 19, 85.
Fixed point arithmetic, 86, 102.
Flag: A 1-bit indicator, 20, 59, 60.
Floating point arithmetic, 10, 78.
Floyd, Robert W, 58.
Footprints, 87, 93.
Fractals, 68, 78.
Fractional precision, 4, 69.
Fragmented fields, 18, 58.
Fredman, Michael Lawrence, 22, 60.
Freed, Edwin Earl, 55.
Frey, Peter William, 94.
Fuchs, David Raymond, 99.
Full adders, 98.
  for balanced ternary numbers, 63.

Gabow, Harold Neil, 95.
Games, 40, 52, 85.
Gaps, between prime numbers, 77.
  between Ulam numbers, 93.
  in a scattered accumulator, 85.
Garbage collection, 27.
Gardner, Martin, 40, 98.
Gathering bits, 83.
Gauß (= Gauss), Johann Friderich Carl (= Carl Friedrich), 36.
Gebhardt, Dieter, 105.
Generating functions, 55, 57.
  Dirichlet, 78.
Gill, Stanley, 11.
Gillies, Donald Bruce, 11.
Gladwin, Harmon Timothy, 8.
Gosper, Ralph William, Jr., v, 56, 70.
  hack, 4, 54.
Graphs, 14–15.
  algorithms on, 27–28, 62–63.
Gray, Frank, binary code, 73, 89.
Gray levels in image data, 59, 67.
Greedy-footprint heuristic, 87, 93.
`GREG` (global register definition), 9, 12.
Grid structure, 36, 98.
Group of functions, 53.
Groupoids, multiplication tables for, 31, 63.
Grundy, Patrick Michael, 71.
Guibas, Leonidas John (Γκίμπας, Λεωνίδας Ἰωάννου), v, 103.
Gulliver, Lemuel, 107.
Guo, Zicheng Charles (郭自成), 41, 65.
Guy, Richard Kenneth, 73, 98.

Hacks, 1–108.
Hagerup, Torben, 88.
HAKMEM, 26, 71, 75.
Half adders, 98.
  for balanced ternary numbers, 63.
Hall, Richard Wesley, Jr., 41, 65.
Hamburg, Michael Alexander, 75.
Hardy, Godfrey Harold, 75.
Harel, Dov (דב הראל), 33.
Heaps, 32.
  sideways, 32, 63–64.
Heckel, Paul Charles, 82.
Herrmann, Francine, 36.
Heun, Volker, 95.
Hexadecimal constants, v.
Hexadecimal digits, 69.
Hobby, John Douglas, 48, 103.
Holes in images, 42–43.
Hollis, Jeffrey John, 62.
Hudson, Richard Howard, 77.
Hunt, Neil, 103.
Hyperbolas, 44, 66, 75, 103.
Hyperbolic plane geometry, 35–39, 47, 64, 97.

Ide, Mikio (井手幹生), 92.
Identities for bitwise operations, 3, 52, 53, 55, 75, 77, 86.
Identity matrix, 108.
ILLIAC I computer, 11.
Implication, in 3-valued logic, 31.
Implicit data structures, 32–39, 63–64.

Independent sets, maximal, 63.
Infinite binary trees, 53.
Infinite exclusive or, 74.
Infinite-precision numbers, 2, 4, 52.
Inorder of nodes, 33.
Inshuffles, 69, 80.
Inside of a curve, 44.
Interchanging two bits, 55.
Interchanging selected bits, 71.
Interleaving bits, 16, 59; *see also* Perfect shuffles, Zipper function.
Internet, ii, iii.
Inverse of a binary matrix, 70.
Inverse of a permutation, 50.
Isometries, 74.

Jardine, Nicholas, 92.
Johnson, David Stifler, 92.
Jordan, Marie Ennemond Camille, curve theorem, 44.

Kaas, Robert, 32.
Kaasila, Sampo Juhani, 102.
Katajainen, Jyrki Juhani, 35.
Kerbosch, Joep A. G. M., 92.
King-neighbors, 40.
Kingwise connected components, 41–43, 65–66.
Kirsch, Russell Andrew, 40, 65.
Knight moves, 63.
Knuth, Donald Ervin (高德纳), i, v, 22, 77, 78, 93, 99, 103, 106.
Kuhn, Markus Günther, 106.

Lakhtakia, Akhlesh (अखिलेश लखटकिया), 75.
Lamport, Leslie B., 19, 20, 59.
Lander, Leon Joseph, 77.
Large megabytes, 77.
Largest element of a set, 11.
Larvala, Samuli Kristian, 83.
Latin-1 supplement to ASCII, 85.
Läuter, Martin, 10.
Lawrie, Duncan Hamish, 81.
Le Corre, J., 13.
Leap year, 88.
Least common ancestors, *see* Nearest common ancestors.
Least significant 1 bit ($2^{\rho x}$), 8–9, 54.
Lee, Ruby Bei-Loh (李佩露), 83.
Left-to-right minimum, 95.
Leftmost bits, 10–11, 22, 55.
Lehmer, Derrick Henry, 4.
Leiserson, Charles Eric, 10, 55.
Lenfant, Jacques, 80.
Lenstra, Hendrik Willem, Jr., 52, 73, 74.
Levialdi Ghiron, Stefano, 42–43.
Lexicographic order, 18, 68.
lg, *see* Binary logarithm.
Life, 40, 65.
Lights Out puzzle, 105.
Linked allocation, 91.
Little-endian convention, 6–8, 12, 20, 28, 76, 77.
Littlewood, John Edensor, 75.
Lobachevsky, Nikolai Ivanovich (Лобачевскiй, Николай Ивановичъ), 36.
Loukakis, Emmanuel (Λουκάκης, Μανώλης), 92.
Lower bounds, 23–27, 61–62, 104.
Lowercase letters, 59.
Lowest common ancestor, *see* Nearest common ancestor.
Loyd, Samuel, 77.
Łukasiewicz, Jan, 31, 63.
Lutz, Rüdiger Karl (= Rudi), 101.
Lynch, William Charles, 11.

Magic masks (μ_k and $\mu_{d,k}$), 9–12, 16, 22, 37, 54, 71, 75, 76, 78–80, 82, 84, 88, 96, 103.
Majority function, 27, *see also* Median function.
Malgouyres, Rémy, 99.
Manchester Mark I computer, 2.
Mann, William Fredrick, 98.
Mapping modules, 58, 81.
Mapping networks, 58, 81.
Mapping three items into two-bit codes, 28–31, 63.
Mappings of bits, 17, 58, 81.
Margenstern, Maurice, 36.
Mark II computer (Manchester/Ferranti), 2.
Martin, Monroe Harnish, 10.
Mask: A bit pattern with 1s in key positions, 9, 16–18, 20, 49, 50, 69.
Masking: ANDing with a mask, 31.
Matrices of 0s and 1s, 67–70, *see also* Bitmaps.
  multiplication of, 50–51, 56.
  transposing, 15, 56, 67, 69, 80.
  triangularizing, 68.
Matrix multiplication, 50–51, 56.
Matrix transposition, 15, 56, 67, 69, 80.
max (maximum) function, 2, 31, 60.
Maximal cliques, 62.
Maximal independent sets, 92.
Maximal proper subsets, 58.
Maybe, 31.
McCranie, Judson Shasta, 93.
Median function, v, 21, 86, 87.
Meerö, László, 105.
Mems: Memory accesses.
Merge sorting, 49.
METAFONT, 103.
mex (minimal excludant) function, 52.
Mikado pattern, 106.
Miller, Jeffrey Charles Percy, 11.
Miltersen, Peter Bro, 27.

min (minimum) function, 2, 31, 60.
Minimal excludant, 52.
Minimum element in subarray, 64.
Minsky, Marvin Lee, 66.
Mixed-radix representation, 60.
MMIX, ii, iv, 5, 7–10, 12, 19, 20, 28, 48–51, 54, 55, 57, 59, 60, 62, 67, 69, 70, 73, 79, 84, 86, 87.
mod (remainder) function, 4.
Modal logic, 31, 63.
mone, 76, *see* -1.
Monotone Boolean functions, 70.
Monotonic portions of curves, 45–47, 66.
Monus operation ($x \dot{-} y$), v, 20, 24, 61, 82, 96.
Moody, John Kenneth Montague (= Ken), 62.
MOR (multiple or), 12, 19, 50–51, 56, 69–70, 86, 94, 103.
Morton, Guy Macdonald, 85.
Most significant 1 bit ($2^{\lambda x}$), 2, 11, 60–62, 89.
MP3 (MPEG-1 Audio Layer III), 51.
Muller, David Eugene, 11.
Multibyte encoding, 68–69.
Multibyte processing, 19–23, 59–61.
  addition, 19, 87.
  comparison, 20–21.
  max and min, 60, 88.
  potpourri, 87.
  subtraction, 59, 87.
Multilinear representation of a Boolean function, 107.
Multiple-precision arithmetic, 6.
Multiplication, 4, 10–11, 22, 61, 78.
  avoiding, 21, 22, 59, 78.
  by powers of 2, 3, 78.
  in Conway's field, 52.
  in groupoids, 31, 63.
  lower bound for, 22, 26, 62.
  of 0–1 matrices, 56; *see also* MOR and MXOR.
  of polynomials mod 2, 70.
  of signed bits, 29–30.
Munro, James Ian, 27.
MUX (multiplex), 50, 83, 86, 103, 107.
MXOR (multiple xor), 50–51, 56, 69–70, 70, 73, 107, 108.
Mycroft, Alan, 20.

Navigation piles, 35, 64.
Nearest common ancestors, 33–35, 64.
Necessity, in 3-valued logic, 63.
Negabinary number system, 52.
Negadecimal number system, 37.
NegaFibonacci number system, 36–39, 64.
Negation, 3, 52, 63.
Nested parentheses, 54.
Newline symbol, 20.
Nicely, Thomas Ray, 77.
Nim, 2, 52.
  addition, 2, 52.
  division, 52.
  multiplication, 52, 73.
  second-order, 52.
Noisy data, 65.
Non-Euclidean geometry, 35–36, 97.
NOT (bitwise complementation), 2.
Notational conventions, v, 81.
  $\langle xyz \rangle$ (median of three), v.
  $u \to^* v$ (transitive closure), 27.
  $\bar{x}$ or $\sim x$ (bitwise complement), 3.
  $x^{\oplus}$ (suffix parity), 55.
  $x \mathbin{\&} y$ (bitwise AND), 3.
  $x \mid y$ (bitwise OR), 3.
  $x \oplus y$ (bitwise XOR), 3.
  $x \ll y$ (bitwise left shift), 3.
  $x \gg y$ (bitwise right shift), 3.
  $x \ddagger y$ (zipper function), *see* Zip.
  $x \dot{-} y$ ($\max(x-y, 0)$), v, 20, 24, 61, 82, 96.
  $(x?\,y\!:z) = xy + \bar{x}z$ (mux), 60, 62.
  $z \cdot\!|\!\cdot \chi$ (sheep-and-goats), 17, 57.
NP-hard problems, 57.
Null spaces, 68.
NXOR (not xor), 79.
Nybble: A 4-bit quantity, 12, 69.
Nyp: A 2-bit quantity, 12, 69.

Objects in images, 42.
Octabyte or octa: A 64-bit quantity, 69.
Odd Fibonacci number system, 96.
Ofman, Yuri Petrovich (Офман, Юрий Петрович), 84.
Omega network for routing, 56–57.
One-to-many mapping, 17, 30.
Ones counting, *see* Sideways addition.
Online algorithms, 42–43, 66.
Open bitmaps, 65.
Optical character recognition, 40, 65.
OR (bitwise disjunction), 2–3.
Ordinal numbers, 73.
Oriented forests and trees, 33, 42.
Oriented paths, 27.
Outshuffles, 56, 69.
Outside of a curve, 44.
Overflowing memory, 92.

Packed data, operating on, 4, 19–21, 31, 59–60, 63, 69.
Packing of data, 4–6, 16, 31, 54, 69, 83.
Page faults, 59.
Paley, Raymond Edward Alan Christopher, 54, 75.
Papadimitriou, Christos Harilaos (Παπαδημητρίου, Χρίστος Χαριλάου), 92.
Papert, Seymour Aubrey, 66.
Parabolas, 44, 66.
Parallel processing of subwords, 19–23, 59–61.
Parenthesis traces, 54.

Parity function, 27, 62, 73, 79.
suffix, 55, 69, 91.
Parity patterns, 67–68.
Parkin, Thomas Randall, 77.
Patents, 79, 83.
Paterson, Michael Stewart, 22, 90, 91.
Pattern recognition, 40.
Patterns, searching for, 20–22, 61.
Pentagrid, 36–39, 64.
Perez, Aram, 51.
Perfect hash functions, 78.
Perfect parity patterns, 67–68.
Perfect shuffles, 16, 50, 56, 57, 69, 80, 88.
Period length, 62.
Permutation matrices, 50.
Permutation networks, 13–15, 57–58, 81.
Permutations,
induced by index digits, 56.
of bits within a word, 13–17, 25, 50.
of bytes within a word, 50.
of the 2-adic integers, 53.
Omega-routable, 56–57.
Perpendicular lines, 36.
Peterson, William Wesley, 51.
Phi (ϕ), 64.
Pi (π), as "random" example, 17.
Pickover, Clifford Alan, 75.
Pigeonhole principle, 104.
Pipelined machine, 48–49.
Pitteway, Michael Lloyd Victor, 45.
Pixel algebra, v, 40.
Pixel patterns, 4, 53.
Pixels, 39–48, 64–68.
gray, 59, 67.
Pólya, György (= George), 75.
Polynomials modulo 2, 57.
multiplication of, 70.
remainders of, 57, 68.
Polynomials modulo 5, 60.
Population count, 11, *see* Sideways addition.
Portability, 7.
Possibility, in 3-valued logic, 63.
Pratt, Vaughan Ronald, 54, 58, 81, 84, 89, 103.
Prefix problem, *see* Suffix parity function.
Preorder of nodes, 33–35.
Presume, Livingstone Irving, 55.
Prime implicants, 63.
Prime numbers, 5, 54.
Printing, 39.
Priority queues, 35.
Pritchard, Paul Andrew, 77.
Prodinger, Helmut, 78.
Program counter, 26.
Projection functions, 9.
Prokop, Harald, 10, 55.

Quadratic forms, 45–47, 66–67.
Quadtrees, 85.
Quantifications, 74, 89.
Queen graph, 92.
Quick, Jonathan Horatio, 53, 58.
Quilt, 4.

Rabin, Michael Oser (מיכאל עוזר רבין), 84.
Radix −2, 52.
Radix conversion, 60.
Radix exchange sort, 91.
RAM (random-access machine), 26–27, 62, 91.
Ramshaw, Lyle Harold, 21.
Randall, Keith Harold, 10, 55.
Randomized data structures, 79.
Range checking, 60.
Range minimum query problem, 64.
Rank of a binary matrix, 68.
Rasters, 39, *see* Bitmaps.
Rational 2-adic numbers, 61.
Ray, Louis Charles, 40.
Raynaud-Richard, Pierre, 106.
Reachability problem, 27–28, 33.
Rearrangeable networks, *see* Permutation networks.
Recurrence relations, 8, 10, 37, 51, 55, 67.
Recursive processes, 15, 17, 32, 52, 72, 84.
Redundant representations, 104.
Reflection of bits, 12–13, 25, 55, 56, 96, 97.
Regular languages, 61.
Reitwiesner, George Walter, 55.
Remainder mod 2^n-1, 11.
Remainder mod 2^n, 4.
Removal of bits, 8.
Replication of bits, 17, 58, 88.
Representation,
of graphs, 27, 62.
of permutations, 57.
of sets as integers, 11, 18, 27–28, 58, 62–63, 75.
of three states with two bits, 28–31, 63.
Reversal of bits, 12–13, 25, 55, 56, 96, 97.
Rhoads, Glenn Charles, 84.
Right-to-left minimum, 95.
Rightmost bits, 8–10, 54.
Rochdale, Simon, 1.
Rokicki, Tomas Gerhard, v, 55, 79.
Rook-neighbors, 40.
Rookwise connected components, 41–43, 65–66.
Rosenfeld, Azriel (עזריאל רוזנפלד), 41.
Rotation of square bitmaps, 67.
Rounding, 33, 86.
to an odd number, 2, 59, 86.
Ruler function (ρx), 8, 20, 21, 25, 26, 28, 32, 53, 55, 60, 64, 78, 100.
summed, 95.
Runlength encoding, 100, *see also* Edges between pixels.
Runs of 1s, 8, 11, 22–23, 55, 61.
Rutovitz, Denis, 40.

S, the letter, 48.
S1S, 74–75.
Saccheri, Giovanni Girolamo, 36.
SADD (sideways addition), 9, 28, 76, 78, 79.
Samet, Hanan (חנן סמט), 85.
Saturated addition, 92.
Saturated subtraction, *see* Monus.
Scattered arithmetic, 18, 58.
  addition, 18, 57.
  shifting, 58.
  subtraction, 58.
Scattering bits, 83.
Schieber, Baruch Menachem (ברוך מנחם שיבר), 33.
Schläfli, Ludwig, 97.
Schroeppel, Richard Crabtree, 26, 52, 82.
Seal, David, 10.
Second-order logic, 74–75.
Security holes, 69.
Segmented broadcasting, *see* Stretching bits.
Segmented sieves, 77.
Sequential allocation, 91.
SET, the game, 93.
Sets, represented as integers, 11, 18, 27–28, 58, 62–63, 75.
  maximal proper subsets of, 58.
Shades of gray, 67.
Shallit, Jeffrey Outlaw, 78.
Sheep-and-goats operation, 17–18, 57.
Shi, Zhi-Jie Jerry (史志杰), 83.
Shift instructions, 3, 19, 52, 61.
  signed, 49, 78.
  table lookup via, 5, 23, 69, 85, 88.
Shift sets, 24–25.
Shirakawa, Isao (白川功), 92.
Shrinking of images, 42–43, 66.
Shuffle network for routing, 56.
Sibling links, 32, 63.
Sibson, Robin, 92.
Sideways addition, 2, 11–12, 55, 62, 79, 94.
  bytewise, 11, 88.
  function νx, 11, 27, 55, 78.
  summed, 55, 82.
Sideways heaps, 32, 63–64.
Sieve of Eratosthenes, 5, 54.
Signed bits, representation of, 29, 55.
Signed right shifts, 49, 78.
SIMD (single instruction, multiple data) architecture, 19.
Simply connected components, 43.
Singmaster, David Breyer, 105.
Six-register algorithm, 103.
Sjöstrand, Jonas Erik, 106.
Slanina, Matteo, 74.
Sleator, Daniel Dominic Kaplan, 4, 98.
Slepian, David, 13.
Smallest element of a set, 11.
Smearing bits to the right, 8, 11, 78.
Sorted data, 54.
Sorting, 60, 75.
  networks for, 58.
Soule, Stephen Parke, 87.
Sprague, Roland Percival, 71.
Squaring a polynomial, 57.
Squines, 48, 66, 102.
SR (shift right, preserving the sign), 10, 49, 76, 78.
SRU (shift right unsigned), 5, 78.
Stanford GraphBase, ii, iii.
Steele, Guy Lewis, Jr., 16, 57, 80, 83.
Sterne, Laurence, iii.
Stockmeyer, Larry Joseph, 81, 84.
Stockton, Fred G., 102.
Stolfi, Jorge, v, 103.
Storage allocation, 22, 54, 59.
Strachey, Christopher, 12.
Straight lines, digitizing, 66.
Stretching bits, 58, 88.
Strings, searching for special bytes in, 20.
Strong broadword chains, 61.
Subcubes, 18, 63.
Subsets, 11, 27–28, 62–63, 75.
  generating all, 18.
  maximal proper, 58.
Subtraction, 3, 52, 59.
  bytewise, 59, 87.
  saturated, *see* Monus.
  scattered, 58.
Suffix parity function, 55, 69, 91.
Sum of bits, *see* Sideways addition.
  weighted, 55.
Surrogates, 106.
Surroundedness tree, 43, 66.
Sutner, Klaus, 105.
Swapping bits, 12–15, 55–56, 107.
  between variables, 71.
SWAR methods, 19–23, 59–61.
SWARC compiler, 85.
Swift, Jonathan, 107.
Sylow, Peter Ludvig Mejdell, 2-subgroup, 74.
Symmetric functions, Boolean, 62.
Symmetric group, 74.
Symmetric order of nodes, 33.

Table lookup by shifting, 5, 23, 69, 85, 88.
Tarjan, Robert Endre, 33, 95.
Ternary vectors, 31.
Tessellation, 36, 47, 64.
Tetrabyte or tetra: A 32-bit quantity, 69.
Thinning an image, 40–41, 65.
Thompson, Kenneth Lane, 68.
Three-register algorithm, 45–48, 66.
Three-state encodings, 28–31, 63.
Three-valued logic, 31, 63.
Tiling, 36, 47, 64.
Time, mixed-radix representation of, 60.
Tocher, Keith Douglas, 2, 59, 85.
Toruses, 65, 87.
Trailing zeros, 8, *see* Ruler function.

Transdichotomous methods, *see* Broadword computations.
Transitive closure, 27, 33.
Transposing a 0–1 matrix, 15, 56, 67, 69, 80.
Transposed allocation, 77.
Traversal in postorder, 95, 100–101.
Traversal in preorder, 94, 100–101.
Treaps, 79.
Triangularizing a 0–1 matrix, 68.
Tricks versus techniques, 2, 103.
Trinomials, 57.
Triply linked trees, 94, 95, 100.
TrueType, 103.
Truth tables, 9, 70.
Tsukiyama, Shuji (築山修治), 92.
Turing, Alan Mathison, 2.
  machines, 98.
Two's complement notation, 2, 26, 71.
Typesetting, 39.

UCS (Universal Character Set), 69.
Ulam, Stanisław Marcin, 93.
  numbers, 63.
Ultraparallel lines, 36.
Unary notation, 60.
Unbiased rounding, 59, 86.
Uncompressing bits, 57.
Underflow mask, 90.
Unger, Stephen Herbert, 19.
Unicode, 69.
Universal Character Set, 69.
Unpacking of data, 2, 4–6, 57, 83.
Unsigned 2-adic integers, 71.
Unsolvable problems, 75.
Upper halfplane, 97.
Uppercase letters, 59.
Urban, Genevie Hawkins, 40.
Uri, Dario, 105.
UTF-8: 8-bit UCS Transformation Format, 69.
UTF-16: 16-bit UCS Transformation Format, 69.

van Emde Boas, Peter, 32.
Van Wyk, Christopher John, 99.
Variance, 57.
Veblen, Oswald, 44.
Vector space, basis for, 74, 106.
Vertex covers, minimal, 63.
Vishkin, Uzi Yehoshua (עוזי יהושע וישקין), 33.
Vitale, Fabio, 35.
Vuillemin, Jean Etienne, 95.

Wada, Eiiti (和田英一), 76.
Warren, Henry Stanley, Jr., 8, 11, 12, 25, 51, 52, 71, 78, 83, 86, 107.
Wegner, Peter (= Weiden, Puttilo Leonovich = Вейден, Путтило Леонович), 8, 12.
Weighted sum of bits, 55.
Welter, Cornelis P., 74, 75.
Weste, Neil Harry Earle, 44.
Wheeler, David John, 11.
White pixels, 4, 40, 67.
Wilkes, Maurice Vincent, 11.
Willard, Dan Edward, 22, 60.
Wilson, David Whitaker, 93.
Wolfram, Stephen, 106.
Wong, Chak-Kuen (黃澤權), 17, 58.
Woodrum, Luther Jay, 77.
Woods, Donald Roy, 85.
Wraparound parity patterns, 67.
Wunderlich, Charles Marvin, 93.
Wyde: A 16-bit quantity, 69.

XL25 game, 105.
XOR (bitwise exclusive or), 2.
  identities involving, 3, 53, 55, 75.

Yannakakis, Mihalis (Γιαννακάκης, Μιχάλης), 92.

Z order, *see* Zip.
Zero-one principle, 54.
Zero-or-set instructions, 9, 10, 88.
Zeta function, 78.
Zijlstra, Erik, 32.
Zimmermann, Paul Vincent Marie, 83.
Zip, the zipper function, 16, 50, 57, 66, 77, 80, 83, 85.
Zip-fastener method, 85.
`ZSNZ` (zero or set if nonzero), 10, 88.
`ZSZ` (zero or set if zero), 9.

ASCII CHARACTERS

	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	
#2x		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	#2x
#3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	#3x
#4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	#4x
#5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	#5x
#6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	#6x
#7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	■	#7x
	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#a	#b	#c	#d	#e	#f	

MMIX OPERATION CODES

	#0	#1	#2	#3	#4	#5	#6	#7	
#0x	TRAP 5υ	FCMP υ	FUN υ	FEQL υ	FADD 4υ	FIX 4υ	FSUB 4υ	FIXU 4υ	#0x
	FLOT[I] 4υ		FLOTU[I] 4υ		SFLOT[I] 4υ		SFLOTU[I] 4υ		
#1x	FMUL 4υ	FCMPE 4υ	FUNE υ	FEQLE 4υ	FDIV 40υ	FSQRT 40υ	FREM 4υ	FINT 4υ	#1x
	MUL[I] 10υ		MULU[I] 10υ		DIV[I] 60υ		DIVU[I] 60υ		
#2x	ADD[I] υ		ADDU[I] υ		SUB[I] υ		SUBU[I] υ		#2x
	2ADDU[I] υ		4ADDU[I] υ		8ADDU[I] υ		16ADDU[I] υ		
#3x	CMP[I] υ		CMPU[I] υ		NEG[I] υ		NEGU[I] υ		#3x
	SL[I] υ		SLU[I] υ		SR[I] υ		SRU[I] υ		
#4x	BN[B] $\upsilon+\pi$		BZ[B] $\upsilon+\pi$		BP[B] $\upsilon+\pi$		BOD[B] $\upsilon+\pi$		#4x
	BNN[B] $\upsilon+\pi$		BNZ[B] $\upsilon+\pi$		BNP[B] $\upsilon+\pi$		BEV[B] $\upsilon+\pi$		
#5x	PBN[B] $3\upsilon-\pi$		PBZ[B] $3\upsilon-\pi$		PBP[B] $3\upsilon-\pi$		PBOD[B] $3\upsilon-\pi$		#5x
	PBNN[B] $3\upsilon-\pi$		PBNZ[B] $3\upsilon-\pi$		PBNP[B] $3\upsilon-\pi$		PBEV[B] $3\upsilon-\pi$		
#6x	CSN[I] υ		CSZ[I] υ		CSP[I] υ		CSOD[I] υ		#6x
	CSNN[I] υ		CSNZ[I] υ		CSNP[I] υ		CSEV[I] υ		
#7x	ZSN[I] υ		ZSZ[I] υ		ZSP[I] υ		ZSOD[I] υ		#7x
	ZSNN[I] υ		ZSNZ[I] υ		ZSNP[I] υ		ZSEV[I] υ		
#8x	LDB[I] $\mu+\upsilon$		LDBU[I] $\mu+\upsilon$		LDW[I] $\mu+\upsilon$		LDWU[I] $\mu+\upsilon$		#8x
	LDT[I] $\mu+\upsilon$		LDTU[I] $\mu+\upsilon$		LDO[I] $\mu+\upsilon$		LDOU[I] $\mu+\upsilon$		
#9x	LDSF[I] $\mu+\upsilon$		LDHT[I] $\mu+\upsilon$		CSWAP[I] $2\mu+2\upsilon$		LDUNC[I] $\mu+\upsilon$		#9x
	LDVTS[I] υ		PRELD[I] υ		PREGO[I] υ		GO[I] 3υ		
#Ax	STB[I] $\mu+\upsilon$		STBU[I] $\mu+\upsilon$		STW[I] $\mu+\upsilon$		STWU[I] $\mu+\upsilon$		#Ax
	STT[I] $\mu+\upsilon$		STTU[I] $\mu+\upsilon$		STO[I] $\mu+\upsilon$		STOU[I] $\mu+\upsilon$		
#Bx	STSF[I] $\mu+\upsilon$		STHT[I] $\mu+\upsilon$		STCO[I] $\mu+\upsilon$		STUNC[I] $\mu+\upsilon$		#Bx
	SYNCD[I] υ		PREST[I] υ		SYNCID[I] υ		PUSHGO[I] 3υ		
#Cx	OR[I] υ		ORN[I] υ		NOR[I] υ		XOR[I] υ		#Cx
	AND[I] υ		ANDN[I] υ		NAND[I] υ		NXOR[I] υ		
#Dx	BDIF[I] υ		WDIF[I] υ		TDIF[I] υ		ODIF[I] υ		#Dx
	MUX[I] υ		SADD[I] υ		MOR[I] υ		MXOR[I] υ		
#Ex	SETH υ	SETMH υ	SETML υ	SETL υ	INCH υ	INCMH υ	INCML υ	INCL υ	#Ex
	ORH υ	ORMH υ	ORML υ	ORL υ	ANDNH υ	ANDNMH υ	ANDNML υ	ANDNL υ	
#Fx	JMP[B] υ		PUSHJ[B] υ		GETA[B] υ		PUT[I] υ		#Fx
	POP 3υ	RESUME 5υ	[UN]SAVE $20\mu+\upsilon$		SYNC υ	SWYM υ	GET υ	TRIP 5υ	
	#8	#9	#A	#B	#C	#D	#E	#F	

$\pi = 2\upsilon$ if the branch is taken, $\pi = 0$ if the branch is not taken